Querying Uncertain Data in Resource Constrained Settings

by

Alexandra Meliou

M.S. (University of California, Berkeley) 2005
Ptychion (National Technical University of Athens) 2003

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Fall 2009

The dissertation of Alexandra Meliou is approved.

Chair	Date
	Date
	Date
	Date

University of California, Berkeley

Abstract

Querying Uncertain Data in Resource Constrained Settings

by

Alexandra Meliou

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

Sensor networks are progressively becoming a standard in applications that require the monitoring of physical phenomena. Measurements like temperature, humidity, light, and acceleration are gathered at various locations and can be used to extract information on the phenomenon observed.

Sensor networks are naturally distributed, and they display strong resource restrictions. Moreover, the gathered data comes in various degrees of uncertainty, due to noisy and dropped measurements, interference, and the unavoidable discretization of the examined domain. A basic task in sensor networks is to interactively gather data from a subset of nodes in the network. Surprisingly, this problem is non-trivial to implement efficiently and robustly, even for relatively static networks.

In this thesis we address the traditional database problem of query optimization in this new setting. We identify the characteristics of sensor network environments and the requirements of applications that are relevant to querying. We focus on making queries more energy efficient by means of minimizing the communication and sensing that is required to provide sufficient answers. Our contributions include theoretical, algorithmic and empirical results. We provide complexity analysis for common data gathering tasks, develop algorithms that approximate the optimal query plans, and apply our techniques to a prototype implementation that tests our theory and algorithms over real world data, demonstrating the feasibility of our approach.

Professor Joseph M. Hellerstein
Thesis Committee Chair

Contents

List of Figures

Acknowledgements

I have been fortunate to always be surrounded by exceptional people, who have helped me aim high. Without all the mentors and friends who have guided me through graduate school, this work would not have been possible. First and foremost, I would like to thank my research advisors: Joe Hellerstein and Carlos Guestrin. I am indebted to Joe for taking me as a Master's student, and guiding me through my first steps in graduate research. I want to thank Carlos for his patience with me, and for being the perfectionist that he is, always pushing me to do better. I consider myself privileged, not only to have been given the opportunity to study at Berkeley, but also to have enjoyed the guidance of these two brilliant people.

I would also like to thank my undergrad advisor Timos Sellis for getting me excited about database research, and also supporting me in my quest to pursue graduate studies in the United States. Thodoris Dalamagas has been a great mentor in my undergraduate research and, along with Timos, guided me through an exciting project, that established the foundation of my future career.

My good friend Alexandros Dimakis, has been a critical part of this achievement, and I cannot thank him enough. He is the one that opened my eyes and showed me a world of opportunities. Without his persistence and enthusiasm, I may have hesitated to take this leap, and my life would not be the same.

I would like to thank my collaborators, David Chu, Andreas Krause and Wei Hong for their hard work, insightful observations and the things I learned from them working together. Also, the whole database group, from my early PhD years to the latest ones: Amol Deshpande, Sirish Chandrasekaran, Sailesh Krishnamurthy, Yanlei Diao, Boon Tau Loo, Ryan Huebsch, David Liu, Fred Reiss, Shariq Rizvi, Shawn Jeffery, Rusty Sears, David Chu, Tyson Condie, Daisy Wang, Eirinaios Michelakis, Kuang Chen, Beth Trushkowsky, Peter Alvaro, Neil Conway. They have made the group a continual source of inspiration and friendship.

Special thanks to La Shana Porlaris, Ruth Gjerde, and the support staff of CUSG, who always did their best to relieve me from administrative and computer trouble.

Thanks to all the Bay Area Greeks: Alex, Eleni, Maria-Daphne, Katerina, Nikos, Dimitris, Manolis, Eirinaios, Theocharis, Tasos, Antonis, Ioanna, Kostas, Tassos, Charis. Also great thanks to Ivan, Jake, and the rest of my friends who have been a family away from home.

Last but not least, great thanks to my sister and my parents who made sacrifices for me, and have supported me throughout the years, even when they disagreed with my choices.

Chapter 1

Introduction

Sensing devices are now used by many practical applications that require monitoring of physical phenomena. Measurements like temperature, humidity, light, and acceleration are gathered at various locations and then distributed and stored in the network of sensors, or transmitted over the wireless medium towards a central location. This data is later used to extract information on the specific phenomenon observed.

Sensing applications pose new challenges to data management research, as these new systems have different characteristics than traditional database systems. Data changes frequently, is naturally distributed across numerable locations with restricted storage and computational capabilities, and communication can be lossy and unreliable. Moreover, the limitations of the underlying equipment and errors in the wireless medium contribute to uncertainty in the accuracy of the data.

Some of the challenges in this new field relate to modeling uncertainty in a way that captures the underlying complexity while keeping the data useful, adapting traditional techniques like querying to account for the limitations of the environment, and ensuring that basic data gathering tasks do not interfere with the network's functionality. This thesis begins by asking the question "how can queries and data gathering be made more efficient in a sensor network?". We explore the limitations of these environments, the characteristics of the phenomena, and the requirements of the applications, to provide a resource-aware solution of a traditional database problem in a completely new setting.

In the following sections we give background on the functionality and applications of sensor networks, focusing on characteristics and issues that affect the handling of data.

1.1 Sensing Devices and Applications

Advances in wireless communication, digital electronics, and micro-electro-mechanical systems (MEMS) technology have enabled the development of low-cost multifunctional sensor nodes of relatively small size, that can communicate with each other in short distances. Sensor networks consist of spatially distributed autonomous sensor nodes, that cooperatively monitor physical phenomena. Using basic measurements, such as temperature, sound, light, acceleration, they can extract information in a variety of environments.

Sensor nodes can be thought of as small computers, very basic in their interfaces, components and capabilities. They commonly consist of a processing unit with limited computational power, limited memory, a variety of sensors, a communication device, and a power source that is usually a battery.

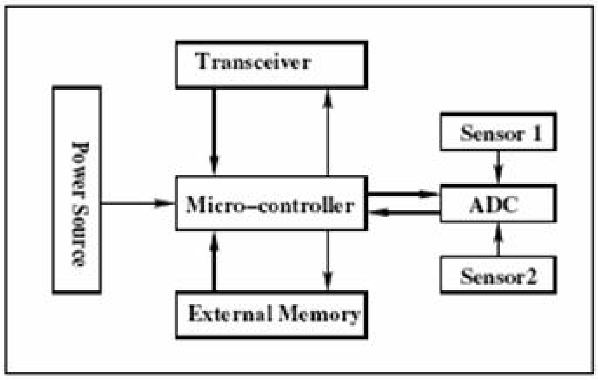


Figure 1.1. A simple sensor node architecture

A network is comprised of a large number of nodes densely deployed inside or close to the monitored phenomenon. A distinguished component of a sensor network is the basestation node, which has access to more computational, communication and energy resources. The basestation node acts as the gateway between the sensor network and the end user or software client.

Depending on the application, sensor networks can present a variety of challenges: limited power, communication failures, mobility of sensors, large scale networks, node failures, ability to withstand environmental conditions, and unattended operation. These introduce many interesting research problems.

1.1.1 Sensing Applications

The study of data management issues in Wireless Sensor Networks has become an important research topic, as sensornets are finding new applications in various domains. In this Section we ground the applicability and relevance of our work, by briefly discussing some example applications of WSNs in different disciplines.

WSNs may have been conceived with military applications in mind, including enemy activity tracking and battlefield surveillance, but civilian applications are now prevalent in many different domains. The advances of technology in remote sensing and automated data collection have enabled higher spatial, spectral, and temporal resolution at a declining cost, changing the field in environmental and biological studies [7], [80]. Deployments in wildlife habitats [60] allow life scientists to put this technology to use, taking advantage not only of the richer data, but also of the elimination of human error and disruption of the natural processes and behaviors under study.

Sensor networks are also used in ecological studies, investigating volcanic activity [81] and endangered species [12], where energy efficiency was identified as one of the main challenges. Other uses have been suggested in environmental monitoring, tracking landfill and air quality [4], controlling chlorine in treated water [5], and monitoring wastewater treatment [3].

Sensing data is also used by *Environmental Observation and Forecasting Systems* (EOFS), which are distributed systems that span large geographic areas and monitor, model and forecast physical processes such as environmental pollution and flooding. Examples include the ALERT [1] and CORIE [79] systems, used to predict future meteorological conditions.

The health industry has also started to introduce sensors for drug administration [6], while there are also some preliminary results on the use of sensors for the health monitoring of cattle [62], by checking the intra-rumenal movement and characterizing the feeding cycle. Several recent projects have also explored the use of sensor networks to monitor the health of buildings, bridges and other structure, an example being the deployment of sensors at the Golden Gate Bridge [47].

In home and business applications, sensors have been used to control energy consumption and offer automation towards a “smart” home/office environment. An example is the “smart kindergarten” [77], designed to assist in early childhood education.

1.2 Energy and Lifetime

In sensing applications sensor nodes are usually battery powered, and in many kinds of deployments batteries are hard, costly, or even impossible to replace. With the exhaustion of their energy source, sensors become unreliable and eventually fail, gradually rendering the network unusable. Therefore, a crucial factor affecting the design and execution of data gathering tasks is a consideration of the energy limitations that are prevalent in sensor network systems.

Depending on the application, the frequency of sensing and querying, and the type of sensors used, battery depletion can occur at different rates. It is however consistent across applications that energy is a crucial resource in the utility of a sensor network. Energy efficiency has become the key parameter in evaluating the behavior of algorithms and components in a WSN, and researchers have tried to tackle it on many different aspects and layers of the system.

In this thesis, we address WSN energy efficiency –and hence lifetime– from the query processing aspect. We argue that queries and data gathering are the most fundamental tasks in a WSN, and the execution of those tasks is the reason why the network was put in place. During query execution and data gathering, three major components come into play: sensing, computation and communication. Communication and sensing are two of the most energy demanding tasks in the function of a sensor node, which leaves a lot of ground for reducing energy consumption by producing smarter query plans that minimize those two components. Given a specific query, an *observation plan* will define the sensors that need to be activated and the communication protocol and paths that should be used to ensure that a sufficient query answer can be constructed. Therefore, when we talk about query optimization in sensor networks, we refer to the construction of optimal observation plans in terms of sensing and communication.

1.3 Contributions

This thesis focuses on the problem of designing efficient query plans in a sensor network setting. We identify communication cost as a central component in energy preservation, and proceed with a theoretical and algorithmic analysis for its minimization. This work can be considered an exemplar of a larger class of emerging challenges, as data becomes increasingly ubiquitous, distributed and uncertain, where the goal is to construct optimized query-specific observation plans, which respect the constraints of the environment; in the case of sensor networks these are energy and communication cost.

We see that the gains in communication can be quite significant in the case of selective data gathering, where only a subset of the network measurements needs to be retrieved. We begin by treating selective data gathering as an independent optimization problem, that can result either directly from selective queries, or indirectly, through other optimization techniques, like using inference to minimize the number of observations. We construct observation plans that minimize the communication cost for those queries, and we also design contingency plans for the case of failures. Our query plans generate paths in the network that gather the appropriate measurements to respond to a specific query. We integrate our routing algorithms with inference techniques, generalizing the optimization problem to account for two parameters: selecting cheap paths in terms of communication cost, and selecting highly informative paths. At this stage planning is performed at a central location using models of the network built from historic data.

Centralized reasoning is better for constructing overall optimal solutions, but in terms of latency and plan robustness, sometimes distributed decisions are more desirable. We therefore develop proactive summarization structures called *in-network summaries* that can be used to make in-network decisions for query propagation and answering. Our distributed approach still uses query specific reasoning, and identifies and solves problems relating to distributed modeling, compression and tree construction.

Finally, we study hierarchical in-network structures for the use in the case of spatially constrained aggregate queries. We construct optimal query plans for arbitrarily shaped regions, and study issues of fault tolerance.

Our main focus is optimizing query plans in terms of communication cost, while using integrated probabilistic models, centralized or distributed, to ensure query satisfaction. This thesis is thematically divided into chapters as follows:

- In Chapter 2, we examine the characteristics and challenges of query routing in sensor networks, talk about approximate queries and the use of inference to optimize data gathering. We describe their characteristics in terms of the data gathered and the types of queries, and specific issues that arise in these settings. We discuss their main restrictions that have inspired the focus of this thesis, and also give some background on model-driven data acquisition, which forms a crucial component in the motivation and further techniques of parts of this work.

- Chapter 3 focuses on the problem of communication minimization as an independent component in the query plan construction. We introduce *data gathering tours* as a novel way to combine query propagation and measurement gathering, and analyze them theoretically. We further introduce and analyze approximation algorithms, proving constant approximation bounds. We keep the analysis grounded via a real world implementation and testing of our algorithms over real data, accounting for practical problems like packet size restrictions. We finally address failures and recovery, which are also tested against real data.

- In Chapter 4, we extend our problem space to continuous queries. We solve the combined optimization problem of minimizing communication and maximizing information, by identifying similarities with the submodular orienteering problem. We further improve our approach with an efficient algorithm that demonstrates gains in both computation times and approximation factor.

- Chapter 5 moves reasoning inside the network. In-network summaries are hierarchical models stored inside the network that can aid in query routing and answering, eliminating centralized planning. The issue of appropriate model compression is central, and we demonstrate how the requirements of the application dictate a specific type of compression that is tuned to the query workload. We further analyze query traversal and hierarchy construction, finishing with a sensitivity analysis over various parameters.

- Finally in Chapter 6 we develop another type of hierarchical summary that is tuned to the answering of spatial aggregate queries. We present query planning algorithms that can deal with regions of arbitrary shape, and analyze fault tolerance.

- Chapter 7 contains discussion of some open questions and some concluding remarks.

Chapter 2

Querying in Sensor Networks

Data collected by sensor nodes must be gathered and processed for the purposes of the application. Usually the role of the collector is bestowed upon a basestation node, which also has the role of forwarding user queries to the network. The query is then appropriately broadcasted to the network, and reaches the destination nodes through a possibly multi-hop path.

The type of queries depends on the application requirements. Sometimes the query can ask for several parameters such as temperature, acceleration and humidity, it may be required to collect and transmit the values one or multiple times, or it may probe for past data to gain statistical information. Our focus will be both on *one-time* queries, and persistent or *continuous* queries. In one-time queries, only the current value of the sensor is needed, whereas continuous queries request the sensor values over a period of time.

Queries can focus on parts of the network (selective data gathering), the entire network, or aggregates of specific attributes. The biggest part of this thesis focuses on `‘‘SELECT *’’` type queries, a term derived from the SQL language convention, referring to queries interested in collecting measurement values from all sensor locations.

With the emergence of sensor networks, a new setting was established where data needs to be manipulated and queries need to be executed and evaluated. *TinyDB* ([55], [57], [59]) was an early software system that offered a declarative interface for sensor network queries, through an adapted version of the SQL standard. Given a query with specified interests, TinyDB collects the appropriate data, filters it, aggregates it and routes it to the basestation node.

2.1 Query Routing

Traditional routing protocols, like that of TinyDB, use flooding to propagate the query to the sensor nodes, and data is then routed to the query location as a separate task. Such an approach makes sense in scenarios where all or most of the nodes need to participate in a query, but can be wasteful when queries target only a small subset of the network nodes. Since the query propagation task does not differentiate between the set of interest and the rest of the locations, these protocols result in all of the nodes participating in the message dissemination. However, communication is a component that consumes a significant amount of battery power, and therefore a non-targeted routing protocol can be very wasteful.

The larger part of this thesis focuses on designing targeted observation plans by combining query propagation and data gathering into a single task. With a query-centric protocol, plans only target the locations where readings are required, avoiding unnecessary messages. We present both centralized and distributed query-centric approaches, where the plans are optimized using probabilistic models of the data.

2.2 Approximate Answer Queries

Sensing applications commonly display a certain tolerance in the accuracy of results. As discussed, uncertainty is often an inherent characteristic of the phenomenon, the methodology or the application, and therefore systems built over such data usually do not expect exact results. At the same time, a deterministic result may not be possible in many applications. Sensing applications monitor phenomena by imposing a discretization to the environment, established by the specific sensor locations. Moreover, faulty sensors as well as lossy communication further contribute to inaccuracies. It is therefore natural that applications will not expect complete accuracy in query results.

A response to a query over data with uncertainty is an estimate of the state of the environment, based on some noisy observations, represented by the probabilistic data values produced by sensors. Estimates are approximate answers which differ from a deterministic answer in that they are accompanied by a qualifier of the accuracy of the response. This qualifier usually represents the confidence in the answer, or the probability that the answer is correct. The accuracy of the answer can refer to the totality of the tuples returned (e.g. *in expectation A% of the tuples are correct, or within certain bounds*), or there can be different confidence returned with every value (e.g. *t1=a with 95% confidence, t2=b with 83% confidence etc*).

With approximate query responses, a variety of new problems surface: is the answer satisfactory? Is it possible to improve on the answer? Whether the accuracy of the answer is sufficient actually depends on the application: weather forecasting would probably be more tolerant to errors than an emergency response system. It is therefore common practice to follow application or query guidelines to decide whether a response satisfies the query. These guidelines are included in the query statement, leaving it to the query planner to construct a plan that produces satisfactory answers. Queries that provide such satisfiability criteria are referred to as *approximate answer queries*.

An approximate answer query specifies an error window and a confidence parameter, which determine whether a response satisfies the query. The smaller the error window parameter, and the higher the requested confidence, the *stricter* the query. Equivalently, the response to an approximate answer query is a set of tuples with a confidence value associated with each value, specifying the accuracy of the answers. The response satisfies the query if the results are accurate enough based on the error window and confidence set out by the query itself.

2.2.1 Exploiting Data Dependencies

Since approximate answer queries specify the accuracy that they can tolerate, probabilistic techniques can be used to improve on the accuracy of results or even make query execution more efficient by cutting down communication cost. This approach is especially applicable in sensor network settings, as the monitored phenomena commonly display strong correlations, and often periodic behavior. For example temperature in a building is expected to be correlated between different locations, and also follow specific variation patterns during the day or the year. Other correlation models in the data can also be used to associate attributes between tuples, or within the same tuple – for example it has been shown that within the same sensor node temperature is correlated with voltage. These correlation models provide a powerful tool in the computation of approximate results.

Correlations can be exploited to optimize query plans for approximate answer queries; in the presence of two highly correlated tuples, it may be sufficient to retrieve only one instead of both of them. In the case of monitoring applications, if the monitored phenomena change at a slow rate, models can be constructed and used to aid in the answering of queries, by reducing the need to access the actual data. These gains can become more significant in distributed settings where bandwidth, latency, and general communication cost can be restrictive. Even in the case when the readings have changed significantly, the models and known correlations can still be useful in determining the new values.

2.3 Model-Driven Data Acquisition

Data correlations have been used for query answering in sensor networks by the BBQ system [28]. Viewing sensor networks as a database ([14], [59]) –a point reenforced by the ability to declaratively query them– can sometimes be problematic, as sensor networks do not exhaustively represent the real world. In a sensornet setting it is impossible to gather all relevant data, as the sensors take samples of discrete points in space. The observations cannot be considered an i.i.d. sample either, as sensor faults, non-uniform placement and packet losses can bias it. Sensornets therefore offer an *approximate* representation of the world, making approximate answer queries suitable for most applications. The traditional approaches to query processing in sensornets ([57], [84]) follow a completist approach, gathering all the available data from the environment, even though most of the data provides little value in approximating answer quality.

Model-driven data acquisition [28] couples data retrieval with statistical modeling techniques, reducing the amount of data that needs to be collected for every query, without compromising answer quality. Statistical models are built and maintained using gathered data, and provide a framework for optimizing the acquisition of sensor readings. For some queries, no acquisition is necessary, if the model itself is sufficiently rich to answer the query with acceptable confidence.

Using models to reduce the cost of data acquisition comes naturally in a sensor network setting, as the physical phenomena measured often display strong correlations and/or periodic behavior. For example, the temperatures of spatially proximate sensors are likely to be correlated, and the temperature variations of a sensor reading throughout the day are likely to follow a common pattern. Given a statistical model over the network measurements, a single sensor reading can be used to improve the confidence of model-driven estimates at nearby locations. Moreover, temporal models can be used to provide current estimates based on older data. With the data gathered by queries, the models are updated, and temporal filters project them to future timesteps. Statistical models can take advantage of spatial and temporal correlations, and also correlations across attributes: for example it is observed that in a sensor node, the voltage is affected by the temperature levels. Measuring voltage is cheaper than measuring temperature, and thus we can optimize the cost of acquisition by electing to measure "cheaper" attributes [28].

The BBQ system [28], which employs model-driven data acquisition, enhances the query processor with a probabilistic model and planner. Models are built using historical data and can be used to answer questions about the current state of the system. The model is denoted by a probability density function, $p(X_1, \ldots, X_n)$, with a variable for each attribute in each sensor. The model is used to estimate the sensor readings at the current time, and these estimates form the query answer. If the confidence in the estimates is not high enough to satisfy the query requirements, the planner can request from the network current readings to improve on the estimates.

The work on BBQ identifies the problem of producing the most efficient query plan as having two aspects. First we want to pick observations that offer the most improvement to the model, and second we want to choose those with the minimal overall cost. To complicate matters, the cost function is not constant: due to multi-hop networking, the cost function is dependent on the nodes already chosen to be queried.

In the next chapter we focus on this topic of solving the data retrieval problem over the non-uniform and dependent cost model that characterizes the sensor network function, which was left unanswered in the original BBQ work.

Chapter 3

The Communication Problem

In this chapter, we consider a basic task in sensor networks: gathering data from a subset of nodes in the network. This problem is posed by model-driven schemes [28], in which an optimization process chooses the set of nodes and sensors to sample in order to approximately answer a high-level SQL query. Note however that it arises in any scenario in which a user or algorithm running at a base station requests readings from an explicit subset of the nodes in the network. The choice of nodes – and the sensors on those nodes – may be made manually based on knowledge of the sensor placement and properties. For example, an office worker planning a last-minute meeting may want to know the sound or light levels in a few specific conference rooms to determine occupancy.

Surprisingly, the problem of interactive data gathering in the sensornet context has not been well studied. The standard approach uses a two-part protocol: query flooding from a basestation, followed by an incast of data from the sensors via a network spanning tree [55]. This approach makes sense in scenarios where all or most of the nodes need to participate in a query. In some cases, however, the set of desired readings is small, and the query needs to be disseminated to only a few nodes in the network; readings are to be acquired at those nodes and returned to a basestation. The combination of flooding and tree-based result routing are ill-suited to these scenarios.

A common concern in wireless sensornet research is that network connectivity is highly unpredictable. However, in many deployments the sensor nodes are fixed in space, and the communication links between the nodes do not demonstrate extreme variation over time – this is the case, for example, in an office environment like Intel's Mirage sensornet testbed [2]. In these cases the network graph can be considered *semi-static*. Although the link quality of an edge demonstrates variations over time, its distribution is practically stationary ([82]). To support that assumption, we analyzed connectivity data from an indoor network of 41 nodes collected every 2 minutes, for a period of 20 hours. Figure 3.1 presents a histogram of the variance of the link qualities. Most links demonstrate very low variance, which shows that the semi-static assumption reasonable.

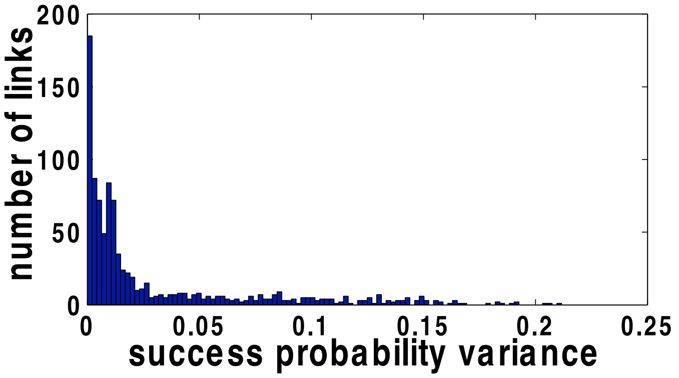


Figure 3.1. Histogram of the variance of the success probabilities of all links.

In such cases the properties of the network links – e.g., the expected number of retries required for pairs of nodes to communicate – can be easily measured by the nodes and periodically propagated to the basestation. By taking advantage of this knowledge, we can develop more sophisticated query routing schemes, where the most efficient communication path is decided at the basestation, which uses source routing to move the query through the network. However, we stress that while the *cost estimates* of such an approach may rest on semi-static properties of the network, the actual routing behavior cannot: transient node and link failures must be handled robustly, even in static deployments in which they are relatively infrequent.

3.1 Related Work

Our work addresses a problem posed in the BBQ query system [28]. In that paper, the authors describe a method of reducing query cost using probabilistic inference. The presented algorithms derive a subset of the network nodes that are sufficient to answer the query within some specified confidence intervals. Our work in this chapter focuses on computing the optimal communication path for retrieving the measurements from this subset. It should not be assumed however that the applicability of this work is restricted to the framework of [28]. Many applications that rely on selective data gathering could benefit from the theory presented in here (e.g., multi-resolution storage [33]). We make the assumption that the basestation possesses information about the entire network topology, which is assumed semi-static. The sensor nodes are not required to maintain any routing information, not even for their immediate neighbors.

A wide range of routing protocols have been proposed for wireless sensor networks, and many of them could be used for selective data gathering. Conventional protocols like flooding or gossip [43] waste bandwidth and energy by making unnecessary transmissions. In a sensornet platform energy restrictions are often very limiting, and the process of data gathering should take energy efficiency into account ([9], [53], [69]). The tradeoff between energy and latency has also been a topic of study ([85]). In this work however we do not include latency as a part of the optimization process. Also, we do not make any assumptions about data correlations as is the case in [22], [23], [70]; if such correlations are exploited, that happens during the node selection that precedes our routing problem [28].

The SPIN protocol proposed in [44] and [48] assumes that all nodes are potential basestations, and the protocol disseminates the data in each node, so that a user posing a query anywhere in the network can immediately get back results. In this scheme, every node is required to know its immediate neighbors, and the protocol does not provide guarantees for the delivery of the data.

In [46] Intanagonwiwat et. al. propose an aggregation paradigm called directed diffusion. This is a data-centric approach that sets up gradients from data sources to the basestation, forming paths of information flow, which also perform data aggregation along the way. Rumor routing [15], [16] also creates paths using a set of long lived agents who direct the paths towards the events they encounter.

More specific to query-centric routing, [52] presents the DIM data structure for embedding indices within the sensor network, to allow more efficient retrieval of events. [57] introduces semantic routing trees, where queries are taken into consideration when the trees are constructed, to facilitate data aggregation. These approaches enable routing by query predicate, rather than by enumerating explicit sets of nodes.

GHTs [68] focus on data centric routing and storage, mapping IDs and nodes to metric space coordinates. One can use GHTs to index nodes by their IDs and achieve a form of query dissemination. We prefer to optimize on the communication cost directly without an intermediate approximate embedding into a metric space.

Since the nodes have no knowledge of the topology, we will propose a packet structure for injecting routing information in the network. This approach makes the problem very similar to the capacitated vehicle routing problem [19], [41], [66]. In capacitated vehicle routing, there exist nodes in a graph that contain an item of a specified volume (analogous to our "measurement set" in Section 3.2). The items need to be picked up by a vehicle (a packet) of a certain capacity and transferred to another node (our basestation). The capacitated vehicle routing problem is to find the minimum cost tours that the vehicles need to make in order to transfer all items. The main difference of this problem with our case is that the packets (vehicles) are required to carry the routing information as well as the data, and packets can be copied mid-tour while vehicles cannot.

We study the algorithmic challenges lurking behind the problem of selective data-gathering in a semi-static sensor network. We define a *base-to-base, source-routed data gathering protocol* that constructs small tours of nodes in the network, starting and ending at the basestation. Each tour combines the tasks of propagating a fixed-size query packet with collecting the requested data: as the query packet progresses through the network, the indicated readings are written into the packet, which eventually returns to the basestation. We achieve our tours via source routing: the basestation uses its knowledge of the network to choose an optimal route for each fixed-size packet, with the final hop of the route being back at the basestation.

While we show that our query-routing problem is NP-complete, we develop polynomial approximation algorithms that produce tours within a constant factor of the optimum. We then enhance the robustness of our initial algorithms to accommodate the practical issues of limited-sized packets as well as network link and node failures, and examine how different approaches behave with dynamic changes in the network topology. Our theoretical results are validated via an implementation of our algorithms on the TinyOS platform and a controlled simulation study using Matlab and TOSSIM [50].

3.2 The Optimization Problem

In our setting we have a semi-static sensornet, and we need to gather data from an explicitly enumerated set of nodes R, which we refer to as the *measurement set*. We assume that there is a powered basestation computer that we will also refer to as the *root* of the network. Querying involves routing a message through the appropriate nodes and receiving the message back at the basestation with the data enclosed.

The network is modeled at the basestation as a graph $G(V, E)$, where V is the set of all nodes and E represents the radio communication links between them. A cost function $c(i, j)$ represents the expected number of transmissions required to send a message over link (i, j). Note that this cost function may not preserve the triangle inequality; while the quality of the communication link is related to the distance between the nodes, it also depends on other features like obstacles that might exist between two nodes.

The cost function is modeled as $\frac{1}{p_{ij}p_{ji}}$, where p_{ij} is the probability that node i will successfully communicate with node j on a given trial. The undirected model ($c(i, j) = c(j, i)$[1].) captures the requirement of receiving an acknowledgement for every message (even if a message is successfully received, the transmission is not considered successful until the sender gets an ack). The same approach was proposed in [82] and [25]. This approach results in an undirected cost graph ($c(i, j) = c(j, i)$), but it does not imply symmetry on the link layer.

The graph model of the network is maintained at the basestation by periodic propagation of link quality measurements. The frequency of such measurements need not be prohibitive in a semi-static network; transient inaccuracies are tolerated by the recovery schemes we discuss in Section 3.6.

Given a network graph G and measurement set R, the optimization problem computes a minimal-cost routing scheme that visits all the nodes in R and brings their data back to the basestation. The communication path can include nodes that don't belong to R and act only as routing nodes, as multi-hop paths can be cheaper than a direct link. The optimization is most naturally solved at the basestation. We therefore adopt a source routing approach, in which the source of the fixed-size query packets (the basestation) marks them with sufficient information to allow nodes in the network to follow the route. In Section 3.5 we elaborate

[1]Asymmetric links are not unusual in the radios of current sensornets, but they can be discarded at the networking layer to avoid unnecessary complexity in routing [42].

on the mechanics of annotating a packet with source-routing information; for our expository purposes in this early discussion we can simply assume that (a) some space in the packet is used to instruct nodes how to acquire data and forward the packet appropriately, and (b) space is available in the packet to store the acquired data from nodes in R as the packet makes its way through the network. Because we use source routing, we do not require nodes to maintain routing or connectivity tables.

3.2.1 Query Dissemination and Answering

Most traditional techniques divide the actions of query dissemination and data gathering into two separate phases. In the scheme that we are proposing, these two phases are combined, and are executed together, along the same communication path.

In the simple circular network graph of Figure 3.2, traditional approaches would require at least eight transmissions to propagate the query and then receive the answers at the basestation node S. In a combined scheme however, the basestation initiates query execution by injecting a message in the network containing sufficient information to route itself along the circular path $S \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow S$. Nodes receiving the message take the appropriate measurements and incorporate them into the message packet before forwarding it to the next node in the path. Integrating query answering with query propagation results in fewer transmissions (just five in our example).

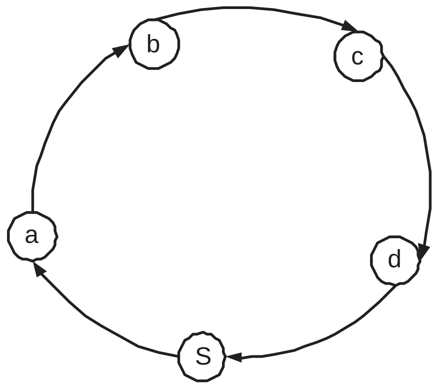


Figure 3.2. Message passing

3.3 Data Gathering Tours

The communication protocol described in Section 3.2.1 produces an observation path represented as a graph $G_s(V_s, E_s)$ where $V_s \supseteq R$ (R the measuring set), and E_s is a multiset of edges $(u, v) \in E$ and $u, v \in R$. The existence of an edge (u, v) in G_s indicates that a message will be sent from node u to node v. Note that G_s is directed, indicating the direction of message passing.

The communication path G_s needs to be appropriately constructed so that it contains paths from the basestation node to all nodes in R which propagate the query to the locations of interest, as well as paths from all nodes in R back to the basestation to ensure the retrieval of answers from all required locations.

More formally, for G_s to be a valid solution to our problem the following conditions are necessary:

- G_s has to span all nodes in R
- G_s has to be connected
- for every node $v \in V_s$ there should exist at least one edge coming to v that can propagate the query to v, and at least one edge leaving v that can return the answers to the basestation [2].

This means that graph G_s needs to be strongly connected, so that it contains a path from every node to every other node. We call a graph G_s with these properties a *Splitting Tour*, in contrast to a traditional graph-theoretic tour which is a simple path that begins and ends at the same node. A splitting tour is a tour that is allowed to split and merge (e.g. Figure 3.3).

The fact that G_s is strongly connected guarantees that all nodes in the communication path are able to both receive the query and deliver the results. A necessary condition for this is that every cut in the graph is of minimum size 2[3]. To see this, first observe that a cut of size 0 would indicate a disconnected graph. Now assume there was a cut (V_A, V_B) of size 1, and suppose the basestation was a node $r \in V_A$, then there would be no way of sending the query to nodes in V_B and retrieving the answers, because of the single edge

[2] This property automatically satisfies connectivity, which was included for clarity.

[3] The size of a cut (V_A, V_B), where $V_A \subseteq V_s$ and $V_B = V_s - V_A$, is the number of edges $(u, v) \in E_s$ where $u \in V_A$ and $v \in V_B$, or $u \in V_B$ and $v \in V_A$.

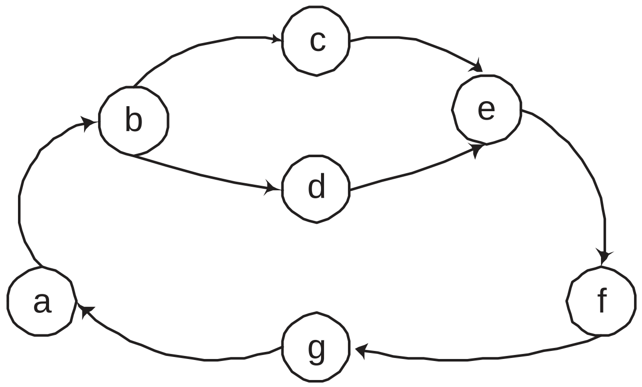


Figure 3.3. A splitting tour, assuming node a as the basestation. The tour splits at node b and follows two separate paths which merge at node e.

connecting V_A and V_B. (Remember that G_s is directed, so using a physical link in both directions counts as two separate edges in G_s.)

The above observation indicates that a necessary condition for G_s to be a splitting tour is that the undirected version of the graph is *2-edge connected*.

Definition 1 (2-edge connected graph) *A graph is 2-edge-connected if the removal of any 1 edge leaves the graph connected.*

Notice however that a splitting tour represents a communication pattern, and as such it should be allowed to use an edge more than once (a node can receive and transmit on the same link). This means that the splitting tour can in general be a multigraph: a graph $G(V, E)$ where E is a multiset, and hence there can be multiple edges between each pair of nodes. We will define a generalization of a 2-edge connected graph which takes this fact into account.

Definition 2 (2-edge connected multigraph) *A 2-edge-connected multigraph is a multigraph $G(V, E)$, where $\forall e \in E$ the graph $G'(V, E - \{e\})$ is connected.*

3.3.1 Routing Rules

2-edge connectivity is a necessary condition for a graph to be a splitting tour, but it is not sufficient. Figure 3.4 demonstrates some examples of 2-edge connected graphs, which however cannot form a splitting tour. Graph 3.4(a) cannot be a splitting tour, because a query can never reach node a, and data from c cannot reach any other node. This is an example of the strong connectivity requirement.

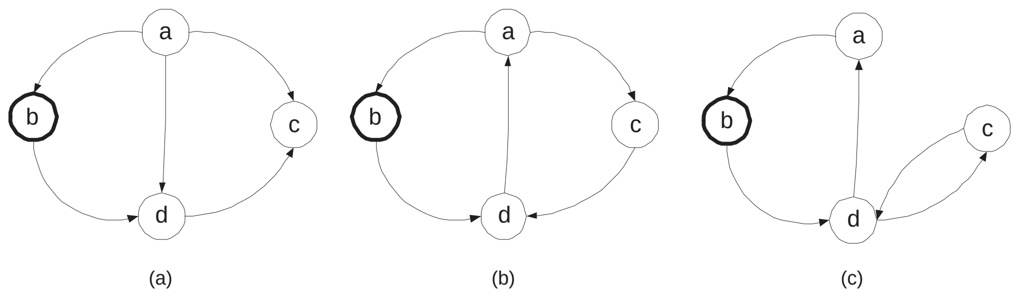


Figure 3.4. Examples of problematic splitting tours (the bold node indicates the basestation).

Graph 3.4(b) shows a more complicated problem. Although the graph is strongly connected, data from node c cannot reach the basestation. Node c can only receive the query after it has traveled through node d, but it has to forward its results through d as well. Without duplication of edges, this graph cannot function as a query plan. This example demonstrates that edge direction in G_s is necessary to define a communication plan.

In Figure 3.4(c) we demonstrate a different case. Node d has two outgoing edges (d, a) and (d, c). For graph 3.4(c) to be a splitting tour, it is necessary that the message is forwarded first along edge (d, c), otherwise data from node c will not reach the basestation. More specifically, node d actually needs to forward the query to c and wait for its data before forwarding along edge (d, a). This example demonstrates the need for an imposed order that nodes follow during message routing. Routing order is defined by *routing rules.*

Definition 3 (Routing Rule) *A routing rule for some node u is of the form $E_{receive} \rightarrow E_{send}$, where $E_{receive}$ is a subset of the node's incoming edges, and E_{send} is a subset of the nodes outgoing edges.*

Node u forwards a message across all edges in E_{send} only after having received messages from all edges in $E_{receive}$

In a splitting tour, for every node with more than one incoming and one outgoing edge we need to have a set of routing rules defining message order over all of its adjacent edges.

In the example of Figure 3.4(c), in order to form a valid splitting tour, node d needs to follow two routing rules: $(d, b) \rightarrow (d, c)$ and $(c, d) \rightarrow (d, a)$.

The order of incoming and outgoing messages specified by the routing rules, also defines a *wait-for* relationship between the nodes.

Definition 4 (wait-for relationship) *A node u* waits-for *a node v, symbolized as $v \hookrightarrow u$, if there exists a routing rule for u for which $v \in E_{receive}$*

Wait-for relationships are transitive, i.e. if $v \hookrightarrow u$ and $u \hookrightarrow w$ then also $v \hookrightarrow w$.

3.3.2 Splitting Tours and 2-edge Connectivity

After defining routing rules, we can give a more clear definition of splitting tours:

Definition 5 *A splitting tour is a directed graph with the following properties:*

1. *It is 2-edge connected.*
2. *It has a node that plays the role of the basestation.*
3. *Every node has a set of routing rules, such that if a message starts from the basestation and follows the routing rules, it will traverse every edge exactly once.*

Our goal is to find the most efficient communication path in the network, that visits all of the nodes in our measurement set. This means that we need to find the graph G_s (splitting tour) with the minimum total cost, as defined by the sum of its constituent edge costs. We made the observation that, by definition, the undirected version of a splitting tour is a 2-edge connected multigraph. As the following theorem states, the converse is also true.

Lemma 1 *G is a 2-edge connected multigraph if and only if there exists a direction of its edges that results in a splitting tour.*

Proof: The fact that a splitting tour is a 2-edge connected graph is given by the splitting tour definition. To prove the converse, i.e. for every 2-edge connected graph, there exists a proper direction of its edges that results in a splitting tour, we need to show that there exists a set of routing rules for every node that gives a valid communication path.

We are given an undirected graph $G(V, E)$. The graph is 2-edge connected, and we choose one of the nodes to be the base station. For every edge $(u, v) \in E$ there exists a cycle in the graph that contains the base station and the edge (u, v), because the graph is 2-edge connected. We can cover all the edges in E with several such cycles. Every cycle is required to contain the source node. See an example at figure 3.5.

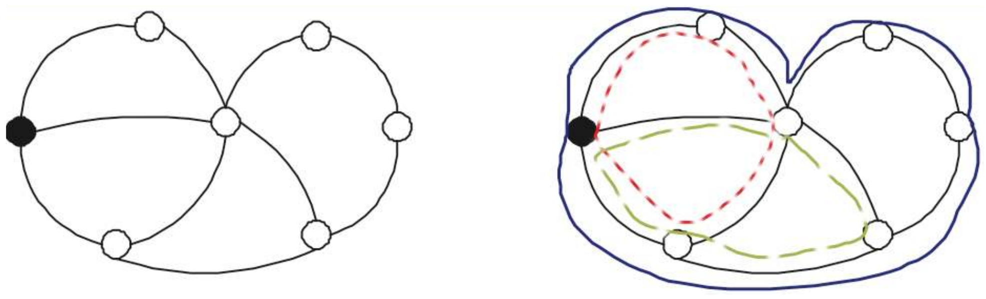


Figure 3.5. Covering a 2-edge connected graph with cycles.

Assume that $S = \{C_1, C_2, \ldots, C_n\}$ is a set of cycles covering all edges of E. We pick the first cycle C_1 and define a consistent direction on all its edges (i.e. we direct all edges by following the cycle from the basestation over all edges in C_1 and back to the source again). It is obvious that for all nodes participating in C_1 a message only needs to traverse every edge of the cycle once to get the data from all of them. We add these nodes to set V_s. V_s contains the nodes for which condition 3 of definition 5 is satisfied. If we make $V_s = V$ then we are done.

We will prove the theorem by induction. In each step we pick the next cycle of S and direct it. Some parts of the cycle may already have a direction because of the previous steps. The parts of the cycle that are left undirected are simple paths, and we will direct them individually. In every step the directed part of the graph must be a splitting tour.

As shown above, this holds for the first step. We assume that after directing k of the cycles of S, the directed part is a splitting tour. We will show that it will remain a splitting tour after directing cycle $k + 1$.

A single undirected path starts from node a and ends at node b ($p = a \to v_1 \to v_2 \to \ldots \to v_t \to b$). If $a \equiv b$ then the path is a cycle and we can choose an arbitrary direction for it. We also need to replace the routing rules accordingly. We randomly pick a routing rule $E_{receive} \to E_{send}$ of node $a \equiv b$ and replace it with the rules $E_{receive} \to (a, v_1)$ and $(v_t, b) \to E_{send}$. We add the nodes of the cycle to V_s, and it is easy to see that the graph is still a splitting tour.

We will now address the case where $a \neq b$. Since we direct every cycle we know that there is at least one incoming and one outgoing edge for both nodes a and b, and because a and b belong to paths that where previously directed, therefore $a, b \in V_s$. If $a \hookrightarrow b$, i.e. b waits for a, then we direct the path from a to b. The new routing rules for the 2 nodes are $E^a_{receive} \to E^a_{send}, (a, v_1)$ for a, and $E^b_{receive}, (v_t, b) \to E^b_{send}$ for b. Since this was a splitting tour before the addition of the path, b could return its data to the basestation. After the addition of the new path, $\forall v \in p,\ v \hookrightarrow b$, so every v in p can return their data back to the basestation.

In the case of $b \hookrightarrow a$, then the process is inverted, and if there is no wait-for restriction between a and b, we can direct arbitrarily. The new routing rules inserted in this last case, will impose a new ordering between a and b. All the nodes of the path get added to V_s.

Every undirected path of cycle C_{k+1} can be directed resulting in a splitting tour. ■

Since we want to find the splitting tour G_s with minimum total cost, from Lemma 1 we see that the problem that we need to solve is equivalent to finding the 2-edge connected multigraph with minimum cost.

3.3.3 Problem Definition

Definition 6 (Minimum Splitting Tour Problem) *Given a graph $G(V, E)$ and a set of nodes $R \subseteq V$, find the minimum cost splitting tour, that goes through all the nodes in R.*

The Minimum Splitting Tour Problem (STP) can be reduced to the following problem:

Find the graph $G'(V', E')$ of minimum weight (minimize $\sum_{e \in E'} w_e$), which has the following properties: $\forall S \subset V'$ and $R \cap S \neq \emptyset$, there must exist at least one edge $e_1 \in E'$ coming out of S, and at least one edge $e_2 \in E'$ going into S.

Even though the graph is undirected, the requirement for an incoming and an outgoing edge ensures that queries can be propagates to all nodes in R and their results can reach the basestation. It is possible that the incoming and outgoing edges coincide, but that would just mean that we need to account for their cost twice, once for each direction.

Now the problem as stated above can be solved by the following linear program:

$$\begin{aligned}
& \mathit{minimize} \sum_{e \in E} w_e x_e \\
& s.t. \\
& \forall S : \sum_{\substack{e=(u,v)\in E \\ u \in S, v \notin S}} x_e \geq 1 \\
& \forall S : \sum_{\substack{e=(u,v)\in E \\ u \notin S, v \in S}} x_e \geq 1
\end{aligned}$$

There is a variable x_e for every edge $e \in E$, and w_e is the weight of this edge. The above linear program can be solved in polynomial time, but gives fractional solutions. We need integer solutions to construct a splitting tour.

3.3.4 Hardness

We now assess the hardness of finding the minimal-cost splitting tour of a graph. We will prove the following:

Theorem 1 *Computing the minimum cost splitting tour of a graph $G(V, E)$ is NP-complete.*

As we know from Lemma 1, finding the min-cost splitting tour is equivalent to finding the min-cost 2-edge connected multigraph that spans all the nodes in the measurement set R. In particular, from now on we will refer to this graph as 2-edge-connected multigraph *embedding*, to emphasize the fact that it is constructed from another graph (G).

The instance of the problem that we are required to solve is the following:

Minimum cost 2-edge connected multigraph embedding (2ECME)

- **Instance:** Graph $G(V, E)$, cost function $c(u, v)$ representing the cost of the edge (u, v), integer B.

- **Question:** Is there a 2-edge-connected multigraph embedding $G' = (V, E')$ of $G = (V, E)$ with $\sum_{(u,v) \in E'} c(u, v) \leq B$?

We will prove that 2ECME is NP-hard. To do this, we will use a reduction from the minimum k-edge connected subgraph problem, which is known to be NP-complete [34]. The minimum k-edge connected subgraph problem is stated as follows:

- **Instance:** Graph $G(V, E)$, positive integers $k \leq |V|$ and $B \leq |E|$.
- **Question:** Is there a subset $E' \subseteq E$ with $|E'| \leq B$ such that $G' = (V, E')$ is k-edge connected?

This problem is NP-complete for $k \geq 2$. From now on, we will concentrate on the case of $k = 2$ and we will refer to this problem as 2EC.

In 2EC, the solution is the spanning 2-edge connected subgraph of G with the minimum number of edges. The difference between 2EC and the 2ECME problem is that the second minimizes the total weight of the graph and allows reuse of edges (i.e., an edge from the input graph can appear twice as 2 different edges in the result).

Using a reduction from 2EC, we can prove the following:

Theorem 2 *The 2ECME problem is NP-hard.*

Proof: To prove this statement we need to demonstrate how an instance of the 2EC problem (which is used for the reduction), can be transformed to an instance of 2ECME in polynomial time. After that, we also need to show that the solution of the 2ECME instance, uniquely defines the answer (yes or no to the decision problem) to the 2EC instance.

Reduction from 2EC:
We are given an instance of the 2EC that has graph $G(V, E)$ and an integer B as inputs. We want to find whether there exists a 2-edge connected spanning subgraph of G with at most B edges.

- Case 1: G has one or more bridges[4]

  In this case, the answer to the decision problem is NO, because there is no way

[4]A bridge is an edge whose removal disconnects the graph.

to construct a 2-edge connected spanning subgraph from a graph that is not 2-edge connected. The existence of bridges can be verified in polynomial time using a modified depth first search.

- Case 2: G has no bridges

  From our instance of 2EC we construct an instance of 2ECME as follows:

  Input: graph $G(V, E)$, cost function $c(u, v) = 1$, $\forall (u, v) \in E$, integer B.

  The output is a spanning 2-edge connected multisubgraph embedding $G'(V, E')$. If $\sum_{(u,v)\in E'} c(u, v) \leq B$ then the answer to 2EC is YES. Otherwise, NO.

We will now explain why the above is true.

If $E' \subseteq E$ (i.e. no edge is used twice), then G' is the actual solution to the minimum 2-edge connected subgraph problem (every edge has weight 1, so the total weight of the graph is equal to the total number of edges used in the 2EC solution).

In the case where G' contains edges that are used twice, we will again prove that the total cost of G' is equal to the number of edges in the output of 2EC.

Lemma 2 *For every edge (u, v) in G', the minimum cut containing this edge, is equal to 2.*

Proof: Due to 2-edge connectivity, the minimum cut will be ≥ 2.

Assume that the minimum cut containing the edge (u, v) is defined by the sets V_1 and $V_2 = V - V_1$, and is of size > 2 [5].

Since G' is 2-edge connected, the minimum cut is ≥ 2. Assume that the size of the minimum cut containing edge (u, v) is > 2. We remove edge (u, v) and get the resulting graph G''. If G'' is not 2-edge connected, this means that the minimum cut (V_1,V_2) is strictly less than 2. Since G' was 2-edge connected, the cut (V_1,V_2) must be ≥ 2 in G'. But we removed only 1 edge, so if the minimum cut containing (u, v) is strictly > 2, then G'' must be 2-edge connected.

This means that we can remove (u, v) from G' and get a 2-edge connected graph of lower cost. So, if the minimum cut containing (u, v) is greater than 2, then G' isn't minimal.

[5]All the nodes in V_1 are connected, because otherwise we could remove the unconnected component and reduce the size of the cut. The same argument holds for V_2 also.

Since G' is the solution to 2ECME, it has to be minimal, and therefore the minimum cut containing (u, v) is 2. ■

Assume that we have an edge $(u, v) \in E$ which appears twice in E' as e_1 and e_2. The minimum cut containing e_1, necessarily contains e_2 as well because u and v will belong to different sets, $u \in V_1$ and $v \in V_2$. Moreover, because of Lemma 2, the minimum cut will contain exactly those 2, and no other edges. In graph G however, both e_1 and e_2 correspond to one edge (u, v). Since G is 2-edge connected, there must exist another edge $e_3 \in E$ in the cut defined by V_1 and V_2 in G, for which $e_3 \notin E'$. We can construct a new graph G'' by replacing one of e_1 or e_2 with the edge e_3, which can be found in polynomial time (they are just back edges from the depth-first traversal of G). The total weight of G'' will be the same as G' because every edge has cost 1 and G'' is the solution to the 2EC problem.

Therefore, the above reduction is valid, and the 2ECME problem is NP-hard. ■

Using Theorem 2 it is now easy to prove Theorem 1.

Proof: (THEOREM 1.) We will use the result of Theorem 2 to show the hardness of the splitting tour problem, and we will also show that the problem is in NP.

Suppose that we could compute the minimum splitting tour G_s in polynomial time. The undirected version of G_s is graph G_u which is a 2-edge connected multisubgraph embedding of graph G. Assume that G'_u is the solution to 2ECME. This means that $|G'_u| \leq |G_u|$. Because of Theorem 1, G'_u can produce a splitting tour G'_s, for which $|G'_s| = |G'_u| \leq |G_u| = |G_s|$. But G_s is minimum, therefore $|G'_u| = |G_u|$, which means that G_u is minimal and can be computed in polynomial time, which is an inconsistency.

Therefore computing the minimum cost splitting tour is NP-hard.

It is also easy to see that the problem is in NP, because it is polynomially verifiable. Given a solution to the optimization problem (the solution would be a set of edges) we can verify in time polynomial to the size of the solution if the answer to the decision problem is positive. The things that we need to check are whether the given set spans all the nodes of interest, whether the result is 2-edge connected (no bridges) and whether the sum of the cost of the edges is $\leq B$; all can be done in polynomial time. Also, to visit any one node, the number of edges that we need to use is no more than $2|E|$, so the solution cannot have size bigger than $2|E||V|$, which means it is polynomially bounded.

Since the problem is in NP and it is NP-hard, it is also NP-complete. ■

3.4 Approximations

Since finding the optimal splitting tour is an NP-complete problem, computing the exact solution is computationally expensive. We need an approximation algorithm that runs in polynomial time. As a first step towards this goal, it is natural to examine the connection this problem has with a very similar graph problem, that is well studied in the literature: the Traveling Salesman Problem (TSP). The TSP produces “simple” tours that do not have any splits, which makes it a special case of the splitting tour. Despite the fact that TSP is also NP-complete, it is a very well-studied problem with many known approximation algorithms, which can give us insight for a solution to our problem.

3.4.1 Bounding the Minimum Splitting Tour with the TSP

Our intention is to provide a polynomial approximation algorithm for the Minimum Splitting Tour Problem (MSTP). We will do that by examining a special case of the splitting tour, which is the solution to the Traveling Salesman Problem. We wish to provide a constant factor bound for our approximations, so we will start by proving that the solution for the TSP is bounded by a constant factor of the solution of the MSTP.

Since the communication path is required to span only the measurement set $R \subseteq V$, we can transform the original network graph to $G_R(R, E_R)$ which contains only the nodes in R, and whose edge set E_R is computed from the original graph G such that each edge $(r, s) \in E_R$ represents the minimum distance path from r to s in G. E_R can be computed in polynomial time by computing all-pairs shortest paths in G. Note that G_R is a complete graph, as long as the original network graph is connected. We will call G_R the *reduced graph* of the network. By definition, since every edge of G_R represents the shortest path in G of the two nodes it connects, the triangle inequality will hold for G_R. The TSP can be solved on G_R and transformed to the equivalent tour in the original graph, by replacing every edge from G_R with the path it represents [6].

The TSP is a special case of the splitting tour, so it follows that the MSTP solution will be at least as good as the TSP solution, i.e., $C^{opt}_{MSTP} \leq C^{opt}_{TSP}$. The question that we need to answer is how much worse the optimal solution of the TSP space will be, compared to the solution from the MSTP space. The answer is given by the following theorem.

[6]Note that we do not mind going through the same node more than once

Theorem 3 *The optimal solution for the TSP cannot be worse than a factor of 1.5 from the optimal solution of the minimum splitting tour problem (MSTP).*

$$C_{TSP}^{opt} \leq 1.5 C_{MSTP}^{opt}$$

Proof: We will show that a tour can be constructed using the edges of a splitting tour graph, in such a way so that the edges used do not exceed in cost a factor of 1.5 of the total cost of the splitting tour graph.

The solution to the MSTP defines a graph $G(V, E)$, which as observed previously is a 2-edge connected graph.

The sum of the degrees of all the nodes V in G is obviously $S(d) = 2|E|$ (every edge is attached to two nodes). $V = V_{odd} \cup V_{even}$. The sum of the degrees of all nodes with even degree is essentially an even number (because it's the sum of even numbers), so $S_e(d) = 2t$. The sum of the degrees of all nodes with odd degree is $S_o(d) = S(d) - S_e(d) = 2|E| - 2t = 2k$. $S_o(d)$ is a sum of odd numbers. But since the result is even, this means that we add an even number of odd numbers, so $|V_{odd}|$ is even, i.e. we have an even number of nodes with odd degree.

Claim 1 *There is always a splitting tour in G that goes through all the odd degree nodes.*

(The above claim is proved after this theorem, as Theorem 4.)

There is a tour T in G that goes through all the nodes with odd degree, and there is an even number of such nodes. Obviously, $C_T \leq C_G = C_{MSTP}^{opt}$, since T uses a subset of the edges in G.

T defines two perfect matchings[7] between the nodes V_{odd}. The length of the tour is the sum of the length of the matches (see example in Figure 3.6).

$$C_T = C_{M1} + C_{M2}$$

We choose a minimum matching M in G for the odd degree nodes. By definition $C_M \leq C_{M1}$ and $C_M| \leq C_{M2}$. This means that:

$$C_M \leq \frac{1}{2} C_T.$$

[7]A matching of the nodes of set V is a set of edges, such that no two of them share a vertex in common. A perfect matching is a matching that touches all vertices.

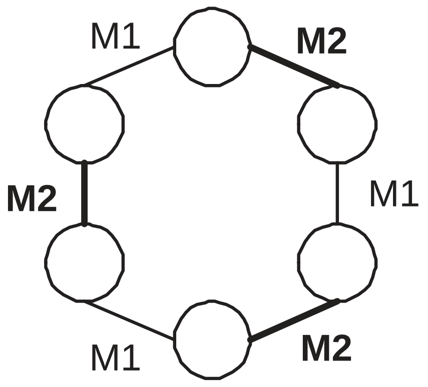


Figure 3.6. A tour T through an even number of nodes defines two matchings between these nodes, M1 (non-bold edges) and M2 (bold edges).

We construct a new graph G' by adding the matching M to G. We know for the total weight of G':

$$\begin{aligned} C_{G'} &= C_G + C_M \leq C_G + \frac{1}{2}C_T \leq C_G + \frac{1}{2}C_G \\ C_{G'} &\leq 1.5C_G \end{aligned}$$

Moreover, all the nodes in graph G' are of even degree. This guarantees that there is an eulerian tour in G'. The solution of the TSP is a tour of minimum length, so it is bounded from above by the eulerian tour in G'. Therefore, we get:

$$C_{TSP}^{opt} \leq 1.5C_G = 1.5C_{MSTP}^{opt}$$

■

Theorem 4 *If $G(V, E)$ is a 2-edge connected graph, then there is always a simple tour in G that goes through all the odd degree nodes.*

Proof: We will show how G can be transformed into a graph $G'(V_o, E')$ where V_o the set of odd degree nodes, E' is a set of edges constructed from E, for which $C_{E'} \leq C_E$, and all nodes in V_o will have even degree in G'. When we construct a graph with only even degree nodes, there exists an eulerian tour in the graph.

In the graph G we are only interested in the odd-degree nodes $V_o \subseteq V$. For constructing a tour through the vertices in V_o, we are allowed (but don't have to) pass through other vertices $V - V_o$, as well. Since we don't care about the other vertices, but can use them for providing paths between vertices of V_o, we can *shortcut* them. By shortcutting a vertex we mean to remove it from the graph, but without removing its adjacent edges. Instead of removing the edges, since the vertex we want to *shortcut* has even degree, we can "pair" its adjacent edges to form new ones and preserve some of the paths that existed before the removal of the vertex. The weight of the new edge will be the sum of the weights of the two edges that created it.

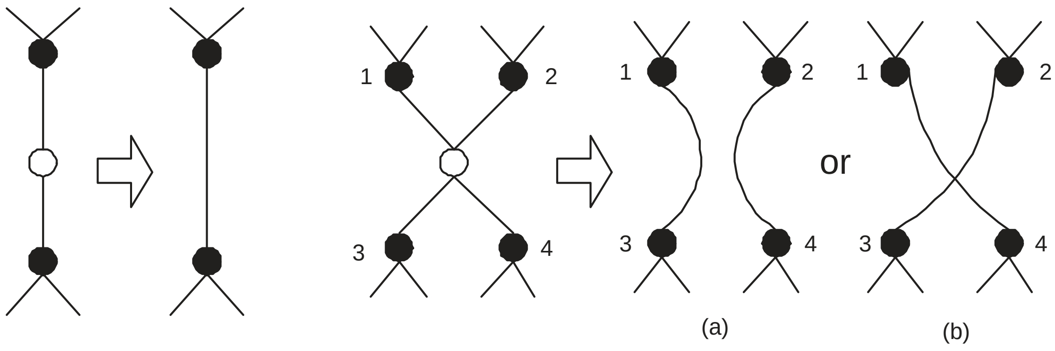


Figure 3.7. Shortcutting even degree nodes.

For 2-degree nodes there is only one way to shortcut them, as shown on the left of Figure 3.7. When the node has higher degree, we can shortcut arbitrarily (see again Figure 3.7), as long as the graph remains connected.

We will show that there will always exist a shortcut that leaves the graph connected, because the graph is 2-edge connected.

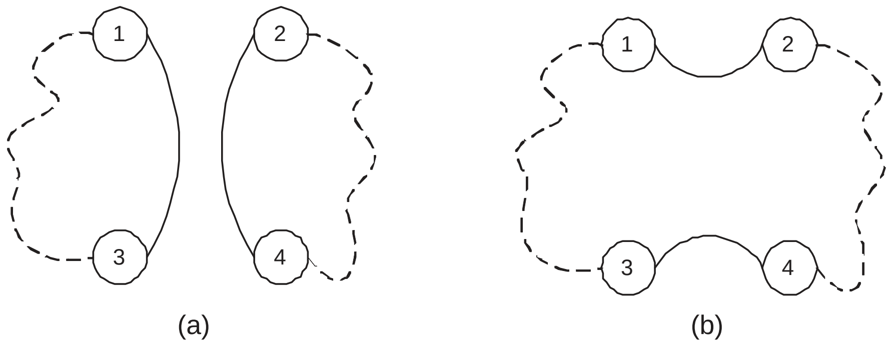


Figure 3.8. Shortcutting 4-degree nodes.

Suppose for example that the shortcutting scheme (a) of Figure 3.8 disconnects the graph. Because of the 2-edge connectivity, there must exist another path (apart from the shortcut) between nodes 2 and 4, and also between nodes 1 and 3. So, by choosing the second scheme, the graph remains connected.

We will now discuss what happens in the case where we want to shortcut a node with an arbitrary even degree.

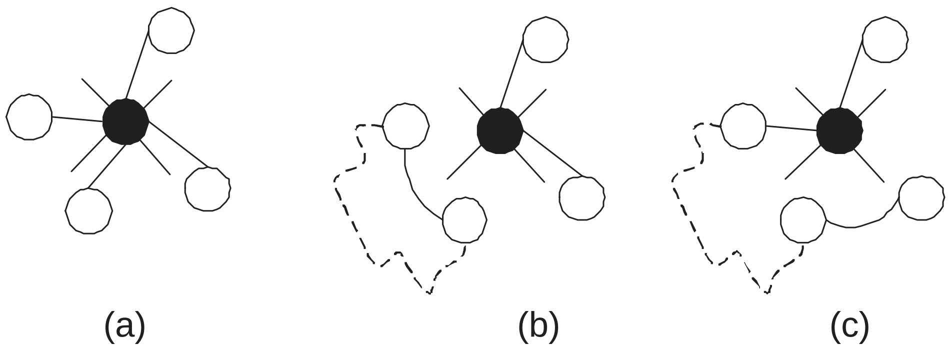


Figure 3.9. Shortcutting k-degree nodes.

Suppose we want to shortcut the black node of Figure 3.9 with degree k. We arbitrarily choose a pair of edges to shortcut (b). If this disconnects the graph, because of the 2-edge connectivity, there is another path apart from the shortcut that connects these two nodes. Therefore, by choosing one of the edges of the pair and shortcut it with any other edge (c) the graph will be connected. So we will end up with a connected graph where the black node that we now need to shortcut has degree $k - 2$. This means that if a $k - 2$ degree node can be shortcutted without disconnecting the graph, then a k degree node can also be shortcutted without disconnecting the graph. So, by induction, we get that for every node with even degree, there is shortcut that leaves the graph connected.

The resulting graph is also 2-edge connected. If it wasn't, then there would exist a subset of nodes for which only one edge went in/out of the set. If that was the case, then the same problem would exist before shortcutting, because we just deleted one vertex, and no edges, so all the subsets of vertices in the shortcutted graph also exist in the original one, with the same edges. This is not possible, because we started with an 2-edge connected graph. Therefore, the resulting graph is also 2-edge connected.

So, at the end of this procedure we get a graph $G'(V_o, E')$, containing only the nodes from graph G with odd degree. Moreover, the total weight of G' is the same as the weight of G (we used the same edges and just got rid of the unneeded nodes). The graph is also 2-edge connected.

G' starts by having only odd degree nodes, but by properly deleting edges we can transform it to a graph that only has even degree nodes:

- For every odd degree node u in the graph, we choose an edge (u, v), which has the property that if it is deleted the graph remains 2-edge connected, and we delete (u, v).

We repeat the above step as long as we still have odd degree vertices in the graph. We will show by contradiction that for every odd degree node there will always exist a proper edge (u, v) to choose in the above procedure.

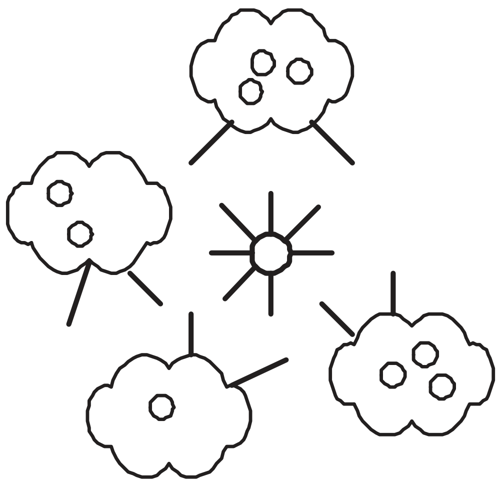


Figure 3.10. If each subset has exactly 2 edges coming out of it, then the total number of edges due to subsets is even, while the edges coming out of the odd degree node is an odd number. Totally we get an odd total number of "incomplete" edges, which cannot be paired.

Assume that there exists a case of an odd degree node v, for which there is no edge that we can delete so that the graph remains 2-edge connected. This essentially means that the rest of the nodes $V - \{v\}$ can be divided into disjoint subsets, each of which is connected to the rest of the graph with two edges, and deleting one of them would violate the properties of the graph (see Figure 3.10 for an example). Node v forms a subset by itself. So every subset has an even degree (number of outgoing edges) apart from node v which has odd. So the sum of all degrees is odd, which opposes the Handshaking Lemma[8]. Hence, such a situation cannot occur.

Therefore, for every odd node in G' there is always an edge starting from this node that we can delete.

[8]Handshaking Lemma: *In any graph, the sum of all the vertex-degrees is equal to twice the number of edges.*

Since this procedure results in a 2-edge connected graph, we can keep applying it as long as we have odd degree nodes. We keep reducing the degree of nodes, till the point where all nodes have even degree. We are definitely going to reach such a state, because in every step we delete edges, and the edges we started with are finite.

Since all nodes have even degree, there exists a eulerian tour that covers all the remaining edges (so it goes through all the nodes). This tour uses edges that exist in graph G, and goes through every $v \in V_o$. ■

3.4.2 A polynomial approximation for the minimum splitting tour

The bound of Theorem 3 does not yet provide us with a good approximation of the minimum splitting tour, because the TSP problem is itself NP-hard. Therefore, we need to provide a bound for a polynomial algorithm, and we will do that for Christofides' approximation algorithm for TSP with triangle inequality[9], which runs in $O(n^3)$ time [21] and produces a result whose cost is at most 1.5 times that of the optimal tour. Since we will use it later on, a sketch of Christofides' Algorithm is presented in Algorithm 1.

Algorithm 1 Christofides' Algorithm for TSP Approximation

```
1: Find a MST (Minimum Spanning Tree) $T_1$. It is $C_{T_1} \leq C_{TSP}$
2: Let $S$ be the set of vertices in $T_1$ with odd degree.
3: Find a minimum weight matching $M$ on $S$. It is proven in [21] that $C_M \leq \frac{1}{2}C_{TSP}$.
4: Construct an eulerian tour $T_2$ on the edges of $T_1 + M$. It will be $C_{T_2} = C_{T_1} + C_M \leq C_{TSP} + \frac{1}{2}C_{TSP} = 1.5C_{TSP}$
```

Now based on the bounds that we proved for the TSP solution, we will prove a constant factor bound for the TSP approximation. It is trivial to show that since $C_{TSP}^{opt} \leq 1.5C_{MSTP}^{opt}$ and $C_{TSP}^{approx} \leq 1.5C_{TSP}^{opt}$, we get $C_{TSP}^{approx} \leq 2.25C_{MSTP}^{opt}$.

However, we are able to prove that the algorithm provides a better bound, as Theorem 5 shows. The proof consists of applying Theorem 3 to every step of Algorithm 1.

[9]Notice that although the triangle inequality does not hold for the original network, it does hold for its reduced graph G_R on which all the algorithms are performed.

Theorem 5 *Algorithm 1 provides a factor 1.75 approximation of the Splitting Tour Problem.*

Proof: For every step of Algorithm 1, we will use Theorem 3, to provide bounds relating to the cost of the splitting tour.

To prove the above statement we need to compare the cost of tour T_2 with the cost C_{MSTP}^{opt} (cost of the splitting tour), instead of the cost of the TSP. We have that:

$$C_{T_2} = C_{T_1} + C_M$$

T_1 is the minimum spanning tree. Assume that T_S is the minimum Steiner tree of a graph G_S representing a splitting tour, which is the tree of minimum cost that spans all the nodes of interest, and can potentially use some other nodes as routing only nodes if the end result is cheaper. Obviously it is:

$$C_{T_S} \leq C_{MSTP}^{opt}$$

It is not straightforward though to compare the cost of the MST on the reduced graph with the cost of the minimum splitting tour. Observe Figure 3.11. The black nodes denote the nodes in set R (nodes of interest). The white nodes do not belong to R, and a tree is not required to span them. Assume that every edge costs 1. The minimum Steiner tree would be the graph itself, with total cost of 3. However, the minimum spanning tree computed on the reduced graph which contains only nodes A, C and D, would be of cost 4, because it would use edge (A, B) twice.

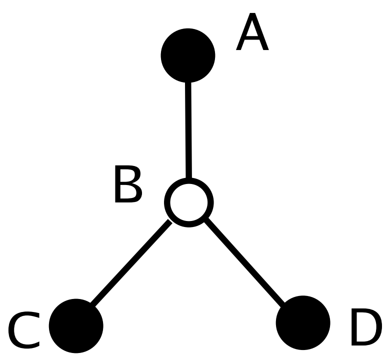


Figure 3.11. Example graph for comparison of the min Steiner tree, and the MST on the reduced graph.

From now on, when we refer to the MST, we refer to the minimum spanning tree of the reduced graph. Also, for simplicity, we will refer to the nodes in R as black nodes and the rest as white nodes. Notice that the MST can also use white nodes, but those will always be of even degree in the resulting tree, participating as intermediate steps in paths between other nodes. The difference with the min Steiner tree is that there the white nodes can be of odd degree. Therefore, obviously, if all white nodes in a min Steiner tree are of even degree, it would be equivalent to the MST.

We will relate the cost of the MST to that of the min splitting tour, by using a transformation of the min Steiner tree. Using the shortcutting technique presented in the proof of Theorem 4, we will shortcut all white nodes with even degree in tree T_S, and we will reduce all those with odd degree, to degree 3 [10]. To this point we have not inflicted any extra cost through this transformation to the cost of T_S. We now have a new tree T'_S where any existing white nodes have degree exactly 3.

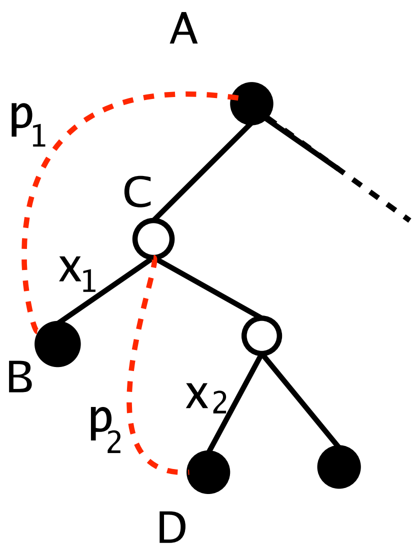


Figure 3.12. Alternative paths between nodes in a Steiner tree.

For every white node in T'_S, there exists an alternative path p_i between the parent of the white node and one of its children, because the graph is strongly connected. For example nodes A and B in Figure 3.12 are connected through path p_1, which is edge disjoint with the path between them already existing in the tree. This result comes from strong connectivity. Also, we name x_i the edge connecting the white node with the corresponding child. Assume

[10]The shortcutting technique will work the same way for the odd degree nodes, as long as their degree is > 3.

that P is the union of all paths p_i: $P = \cup_{i=1}^{k} p_i$. This means that P contains a set of edges and all of the nodes A, B, C, D etc. If $Cost_P < \sum_{i=1}^{k} x_i$, then T_S would have chosen P instead of the x_is. But T_S is minimum. This means that $Cost_P \geq \sum_{i=1}^{k} x_i$.

Therefore we can duplicate all edges x_i making every white node to have degree 4. The cost of the resulting structure will be $Cost_{T_S} + \sum_{i=1}^{k} x_i$, which is less than the cost of the minimum splitting tour because P is contained in G_S. So, the white nodes can now all be shortcutted, resulting in a tree T_1' spanning only the black nodes.

Therefore, we get for the cost of the minimum spanning tree in the reduced graph:

$$C_{T_1} \leq C_{T_1'} \leq C_{MSTP}^{opt}$$

As proven by the Christofides algorithm, it is $C_M \leq \frac{1}{2} C_{TSP}$, so from Theorem 1 we get:

$$C_M \leq \frac{1}{2} C_{TSP} \leq \frac{1}{2}\frac{3}{2} C_{MSTP}^{opt} = \frac{3}{4} C_{MSTP}^{opt}$$

Therefore

$$C_{T_2} = C_{T_1} + C_M \leq C_{MSTP}^{opt} + \frac{3}{4} C_{MSTP}^{opt} = 1.75 C_{MSTP}^{opt}$$

■

Therefore, using Algorithm 1 we can compute in polynomial time a simple tour which we know cannot be more expensive than 1.75 times the cost of the actual optimal solution of our routing problem.

3.5 Practical Considerations

The previous section established a theoretical basis for our problem. However, we have yet to handle a number of important practical considerations. The first, which we address in this section, is the fact that radio network packets are of small fixed size, and source routing instructions for long tours may not fit in a single packet. We begin by describing the specifics of our packet routing implementation in Section 3.5.1, and then three schemes for dealing with long tours in Sections 3.5.2 through 3.5.4.

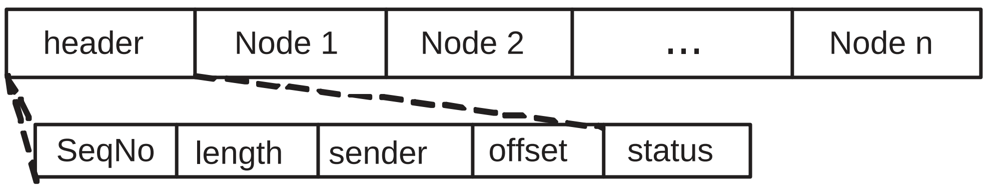


Figure 3.13. Packet structure.

3.5.1 Path injection

In the beginning of this Chapter, we discussed in general terms the idea of source routing in a sensor network. Here we provide more detail. We use the simple packet structure shown in Figure 3.13. The packet header in our implementation includes a sequence number which gets incremented as the packet gets routed around the network, a field indicating the total number of bytes being sent, an offset field to ensure proper packet ordering, the ID of the sender, and 2 bytes with information on the status of the route (mainly used for recovery).

The main part of the packet represents a simple path of size n, which should be traversed in the order indicated, from node 1 to node n. Every node in the path is given a slot (in our implementation 2 bytes are assigned to each slot), which serves as the storage space for all the information that needs to be sent to the network. A typical slot entry includes the nodeID, the ID of the sensor to be used, and the maximum number of retries to be attempted for the next hop.

In our implementation, the packet works as a moving cyclic buffer. When a node receives the packet, it removes itself from the beginning and shifts everyone to the left by one slot. Whenever a node in the path receives a message, its node ID should be in the first slot of the packet. If the node needs to return a measurement it will add it at the end of the packet. If not, it needs to take no further action than forward the message to the next node in the path whose ID is now placed in the first packet slot. Following this procedure, when the packet comes back to the basestation, it will contain the measurements in the same order as the tour was traversed. This packet structure serves both as a command and as a storage medium, and in order to discriminate between the routing and the measuring part of the packet, special delimeters can be used. An example of how the packet gets routed is given in Figure 3.14.

The advantage of using this dynamic scheme compared to a static one-slot allocation per

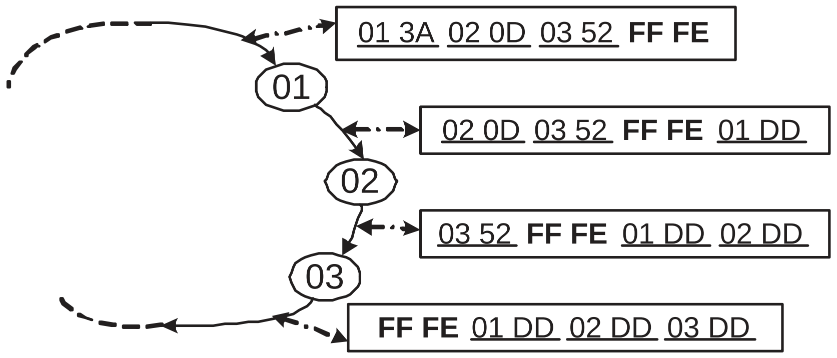


Figure 3.14. Example of how the packet changes from hop to hop. Two bytes are allocated per node. The first one represents the nodeID and the second holds the necessary data to instruct the node whether it needs to sample or not, how many retries it should attempt for the next hop etc. A byte with the value `0xDD` in the figure represents sampling data stored by the corresponding node in the packet. The bytes filled with the values `0xFFFE` are special delimeters that separate the routing information from the data storage.

node in the path, is that routing-only nodes – i.e., those not contained in the measurement set – get completely removed after being visited, making the packet shorter. This feature can improve performance for some traversal methods, as we discuss in Section 3.5.3.

3.5.2 Cutting a tour

Given that background on our path injection scheme, we can now consider the problem of routes that do not fit in a packet.

Suppose for example that we have computed an approximate TSP communication tour T_G on the measurement set using Algorithm 1, with T_G consisting of 15 nodes beginning and ending at the basestation. Assume we have room for 20 bytes worth of slots in each packet, and each slot takes up 2 bytes. We would need 30 bytes to represent T_G, which clearly cannot be done in one packet.

Suppose that a packet can hold tours of maximum size P, i.e., that the packet has enough space for P node slots. One approach is to cut the tour T_G into smaller parts, each of which will have maximum size P. Cuts in the tour are going to be performed by re-routing intermediate edges to the source.

Algorithm 2 uses dynamic programming (DP) to compute the optimal cutting of a long tour T_G into smaller tours of size $\leq P$, so that each one can fit in a single packet. If we assume that the nodes in the network know the ID of the basestation, we can exclude

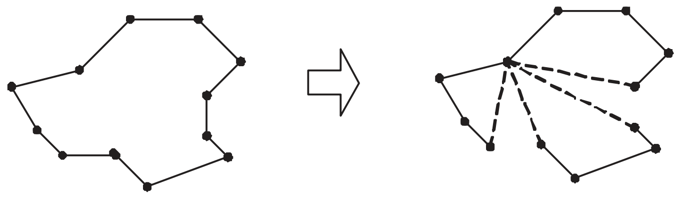


Figure 3.15. Cutting a tour into smaller subtours.

the basestation from the packet as an optimization, but this is not a requirement for the algorithms described. The algorithm computes a *cost-to-go* function, $J(i)$, that represents the cost of the best cutting of the segment from the ith node to the end of T_G. At completion, $J(1)$ will hold the best possible cost of cutting T_G into parts of maximum size P. We can obtain the optimal cuts with another pass over J in the usual DP fashion.

Algorithm 2 Cutting a tour

```
for $i = 1$ to $n$ do
  $J(i) = \infty$
end for
$J(n) = L(n, n)$
for $i = n - 1$ to 1 do
  for $j = 0$ to $P - 1$ do
    if $i + j + 1 \leq n$ then
      //check for path length
      $J(i) = \min(J(i), L(i, i + j) + J(i + j + 1))$
    end if
  end for
end for
```

At the core of the computation is the *local cost function* $L(i, j)$, representing the minimum cost tour that fits in a packet of size P and visits the subset of nodes of T_G that are in the segment from i to j:

$$S \to \ldots \to i \to \ldots \to j \to \ldots \to S.$$

The local cost function $L(i, j)$ can also be computed efficiently: We first precompute a hop-restricted distance function, $d(u, v, k)$, representing the cost of the shortest path from

u to v that uses at most k hops. This function can be computed by a standard DP. $L(i, j)$ is then obtained by another DP that iterates through the nodes of T_G in the range $[i, j]$, using $d(u, v, k)$ as the local cost function.

Note that this algorithm does not modify the order in which the measuring nodes (nodes of T_G) are visited, but the paths followed in between may differ. There are cases where the algorithm may not do any cuts at all, and just change the paths followed. For example, consider the tour $T_G = S \to a \to B \to c \to d \to E \to f \to S$, and the packet size $P = 4$. S is the basestation and the nodes in capital letters form the measurement set $R = \{B, E\}$. T_G does not fit in one packet, so we run the cutting algorithm. It is possible for the output to be a single tour, e.g., $S \to a \to B \to g \to E \to S$. This tour may be more expensive than T_G, and that is the reason it may not have been chosen the TSP approximation, but it does fit in a single packet. Another possible output could be $S \to a \to B \to c \to S$ and $S \to d \to E \to f \to S$, where T_G was divided into two smaller tours by simply short cutting to the root at nodes c and d. Each of these shorter tours fit in a packet. The cutting algorithm will dynamically pick the cheapest of the possible choices.

The cost of the main DP algorithm is $O(nP)$. Additionally, we must consider the cost of precomputing the local cost function L. The subfunction $d(u, v, k)$ is computed in $O(n^2P)$. The L function itself is computed in $O(n^3P)$. Thus, the total cost of the cutting algorithm is $O(n^3P)$.

3.5.3 Multiple packets

As an alternative approach to cutting a tour that cannot fit in a single packet, one can use a "train" of multiple packets to inject the path to the network. We have implemented this approach by adding two fields in the packet header that indicate the total length of the packet-train and the current packet's offset in the train. Upon receiving all packets of the train, a node can reconstruct a virtual "big" packet containing the whole path, process it, break it up again into a train and forward it. Notice that even if for some reason packets arrive in a different order, we can reconstruct the proper order by the header information. In every step we treat the packet-train as one big packet.

One thing to note is that by using the policy described in Section 3.5.1, a packet-train can become shorter while getting routed on the path, because of the removal of the routing-only nodes.

Each packet only needs to provide routing information for one part of the path, but (in the absence of any in-network routing information) all the packets need to be routed through the whole path in order for the measurements to find a path to the basestation. Notice that with this approach the edges of the tour are traversed multiple times, as multiple packets are sent over each edge. The measured cost for every edge will be proportional to the number of packets that use that communication link.

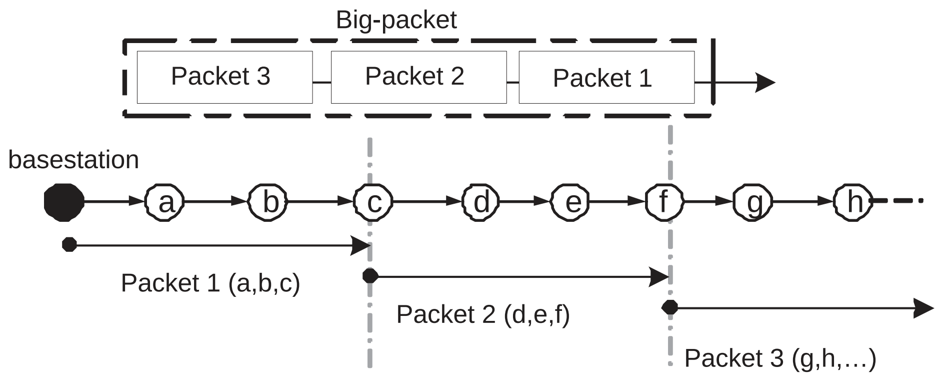


Figure 3.16. Every individual packet holds information for some part of the route. All of them combined can behave like one big packet that holds the whole path and traverses it. Note that during hops data gets transferred between one packet to another, because they all together form a big cyclic buffer.

3.5.4 Hybrid: cutting with multiple packets

Simple cutting of a tour does not allow us to reach nodes that are more than P hops away, and forces us to take a small number of (potentially) expensive edges when collecting data from faraway nodes. Multiple packets, on the other hand, can reach faraway nodes, but may be wasteful when collecting data from a large number of nodes. In this section, we use dynamic programming to combine the strengths of these two approaches.

This hybrid algorithm is similar to the cutting DP procedure in Algorithm 2, but instead of restricting cuts to be of length P, a cut can now have length up to n. We must also modify the local cost function $L(i, j)$ to allow for the use of multiple packets to visit the subset of T_G that is in the range $[i, j]$. A simple approach for computing the multiple packet version of $L(i, j)$ is to first run the algorithm we used in Section 3.5.2, setting the packet size to P; then, we run the same algorithm with a packet of size $2P$, computing the edge costs accordingly[11], then for $3P$, and so on. Finally, we define $L(i, j)$ to be the minimum over all of these packet size options.[12] The final computational cost of this algorithm is $O(n^5)$. The paths obtained by the two previous approaches are strictly more costly than this one, since the hybrid algorithm finds the optimal cut that could use one or more packets per section. In Section 3.7 we will assess the merits of the various schemes in practice on a real network graph.

3.6 Failures

Having dealt with the practical issue of finite-sized packets, we now turn our attention to a subtler practical issue: the dynamics of real networks. For purposes of route selection we assumed that the network is semi-static, but connectivity changes do occur in wireless sensor networks. Link qualities can change and nodes can fail. We want our data gathering approach to remain robust in the face of these events, even if we expect the dynamics of the network to be relatively modest.

In the graph that models the network, the cost of an edge is calculated as $\frac{1}{p_{ij}p_{ji}}$ where p_{ij} the probability that node j receives a packet transmitted by node i. Therefore, the cost of a link gives an estimate of the number of retries a node has to do in order to successfully transmit a packet over this edge and receive an acknowledgement.

If after a certain number of retries specified by the quality of the link – a bad link means more retries are required – a node wasn't able to successfully transmit a message, it has to assume a failure of either the link, or the node it wants to transmit a message to.

To resolve failures we propose two different schemes, backtracking and flooding based recovery.

3.6.1 Backtracking

In this section we will describe the recovery technique of backtracking. Since the nodes do not have any knowledge of the network topology, if the path they are given fails, the

[11]For every path we know which nodes are routing-only and will be removed, so we can pre-compute how long the packet train traversing a specific edge will be. Then the cost of that edge is calculated as the basic cost of transmitting one packet times the number of packets in the train.

[12]We can also use a modified version of the DP algorithm to compute the multiple packet version of $L(i, j)$ more efficiently.

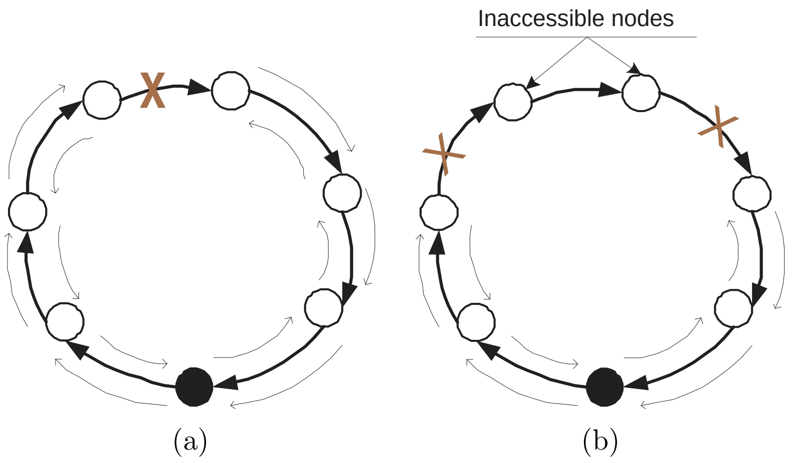


Figure 3.17. The bold edges indicate the initially computed tour. (a) During the traversal a failure is encountered and the message backtracks to the root; a new message is issued in the opposite direction than the tour was defined to gather data from the unvisited part. (b) In case of multiple failures nodes can become inaccessible.

simplest thing they can do is trace back their steps. When a node encounters a failure, it initiates backtracking which will send the message back to the root with as much data as it has gathered, in the same path that it came from. The information needed for backtracking can temporarily be stored in the network: upon the arrival of a message, the receiving node can store the ID of the sender, just for the duration of the query execution, and then, during backtracking, nodes can use this "breadcrumb" information to traverse the path backwards. When the basestation receives the backtracked message, it can issue a message in the opposite direction of the original tour, to attempt to reach the nodes that were missed in the first round-trip. An example is presented in Figure 3.17(a).

Notice that in the case of multiple failures happening in a single tour, some of the nodes may remain unvisited like the example in Figure 3.17(b). In this case, the user who issued the query can be notified about the missing measurements. If this is not acceptable, a new tour can be computed for the missing nodes taking into account the information about the failures the previous run encountered. Notice however that in the case of failures there is no guarantee that we can gather all the data that we intended to gather. It is possible that link or node failures may have disconnected the graph, or some nodes that we needed to take measurements from might have been among the failures.

The backtracking algorithm that we presented is a simple heuristic to handle a small number of failures in the system. It offers full recovery for single failures per tour, but cannot retrieve nodes that fall in between failures in a path. Notice however that the communication cost of performing recovery with backtracking is bounded by a factor of 2 from the cost of the tour, because every edge of the tour will be traversed at most twice (one during forward processing and one during backtracking).

We note that our description of backtracking assumes that there will be no failures during the backtracking step itself. If a message has traveled on a path from the basestation to node n_i and then encountered a failure, backtracking assumes that the message can follow the opposite path, from n_i to the basestation without failures. This is not an unreasonable assumption because the running time of a query performing this type of data gathering will be relatively small and the probability of a failure happening within this interval will be low. Obviously in the event of such a failure, the basestation receives no data from that packet, nor any information on where in the network the packet was lost. After a suitable timeout it can re-attempt the packet either directly, or in the reverse direction.

3.6.2 Flooding

Another approach to recovery is to perform local flooding in case a failure is detected. When a node A exhausts the number of retries denoted in the packet, it will enter recovery mode and broadcast the message in the hope that some node in the unvisited part of the path will hear it. The flooding message contains a TTL (Time To Live) number, which determines the depth of the flood. Upon reception of a flooding message a node B examines it to check whether it is itself part of the path or not. If it is not, it will continue the flood, decrementing the TTL. Flooding terminates if TTL reaches 0.

If B is a member of the yet unvisited part of the path, it can make different decisions as to what it should do with the packet. It can either start sending the packet forward in the path, or send backwards to retrieve any measurements that may lie between nodes A and B, or wait to see if a forwarding message will come from some node preceding B in the path. An example of flooding based recovery is demonstrated in Figure 3.18.

The more specific semantics of the flooding based recovery scheme in the way that we actually implemented it, taking a conservative approach, are described in the following list.

- During normal execution, a node sends only forward
- When a forward sending fails (after specified retries), a recovery bit is set in the

packet, and the node broadcasts the packet. The initiator of the flood is A and the part of the path that is still unvisited is P.

- When a node B hears the flooding message, if B is not in P, and $TTL > 0$, then B continues the flood.
- If $B \in P$, and B heard the flood or a backtracked message during recovery:
  - If no measuring node exists in the path interval (A, B) then B resumes forward execution.
  - Once B has sent a normal-case, forward-directed message, then B will never backtrack.
  - If B has already backtracked then B takes no action for the new flooding or backtracking message (i.e., backtrack only once).
  - If there exists a measuring node in interval (A, B), and B hasn't heard a forward message and B hasn't already backtracked, then B backtracks.
  - If a backtracking message fails (after a specified number of retries) then B resumes normal execution by sending a message forward.

For the last two points of the above list, we chose to follow a conservative approach targeted to the retrieval of as many measurements as we can, without trying to optimize the cost. For example we could possibly have less transmissions if B waited instead of instantly backtracking, because someone else preceding it may have heard the flood and already initiated a forward execution. This would on average decrease communication cost, but it would increase the latency. In this space there is some room for further investigation of the tradeoffs of these parameters, and how they affect the recovery scheme.

Compared to backtracking, a flooding based scheme has the advantage that it can recover from more than one failure in the current tour. A disadvantage however is that the cost (number of messages sent) is not theoretically bounded by a constant factor and depends on the network topology. Also, TTL is a parameter that affects both the cost and the recovered measurements.

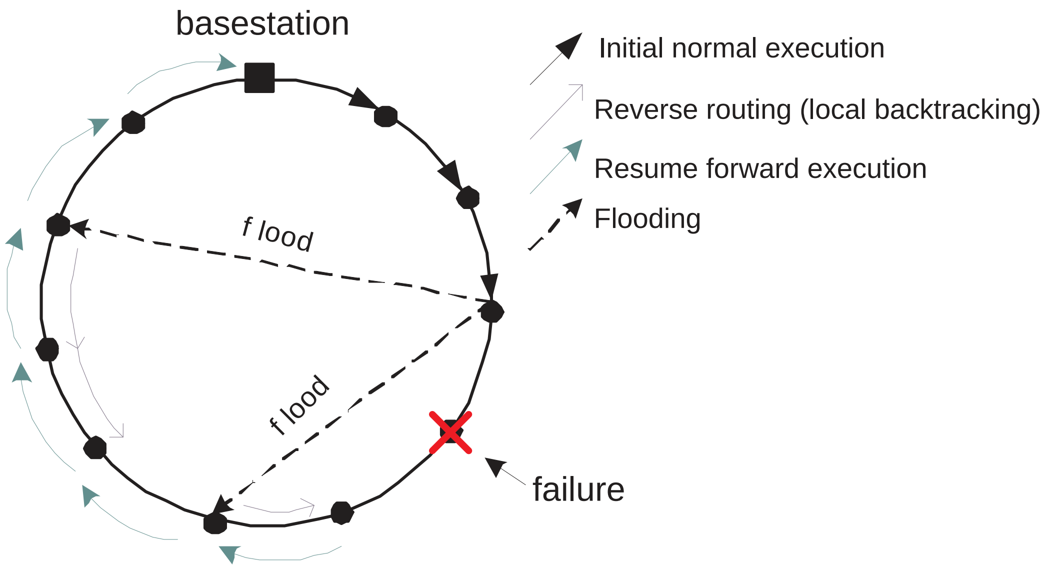


Figure 3.18. When a node detects a failure on the path it initiates a flood with small depth, so that it will remain local. The nodes in the unvisited part of the path that hear the flood backtrack on the path to get any data possible between the failure and their position. If a forward and a backtracking message meet, the backtracking one is killed.

3.7 Experimental Results

We evaluated our proposed schemes via an implementation, that consists of two separate components. The first involves several Matlab routines used to perform the optimization described in Section 3.4, as well as to apply the packet size restriction of the network, using the algorithms presented in Section 3.5. Each tour is stored in a file which is subsequently sent to a Java interface that can parse it and inject the proper packets into the network.

On the network side, our mote code is written in nesC, on the TinyOS platform. This code implements the proper handling of the routing messages, as well as the two different recovery modes, backtracking and local flooding.

For our simulation experiments we gathered connectivity data from a real deployment, through TinyDB queries running for a number of epochs. The connectivity data was given as input to the simulations, thus modeling the dynamics of the real network. We chose to use the public mote testbed at Intel Research Berkeley, which is remotely available via the Mirage resource allocation system [2]. At the time, the testbed consisted of 96 Mica2 nodes at fixed positions. The environment of the deployment is relatively noisy, and includes human activity as well as other radio traffic (802.11, cordless telephone headsets, cellular phones, etc).

3.7.1 Simulation Results

The results that we present in this section consist of two kinds of simulations. The first are Matlab simulations of the network and algorithms, the purpose of which is to provide an insight as of the behavior of the algorithms under different packet requirements. The second class of experiments uses the actual NesC code for the protocols, but instead of running them on the live testbed we ran them within the TOSSIM simulator, which simulates a network of TinyOS motes [50]. We focused on TOSSIM rather than the live testbed in order to be able to control our experiments and validate their behavior.

Experiments were performed by picking random subsets, as the measuring set, from the real network graph and computing the approximation of the optimal solution proposed in Section 3.4. The main goal of our analysis is to compare the heuristics that we proposed in Section 3.5, as well as evaluate our recovery algorithms in the cases of failures.

The reason for introducing the cutting and hybrid heuristics was to address the packet size limitations that are present in a real sensor network environment. Therefore we want to assess their behavior for different packet sizes. Figure 3.19 demonstrates how the communication cost of the different algorithms converges rapidly to the optimal as the size of the packet increases. We have performed the same experiment for different network sizes, as well as different sizes for the measuring set. All the results of these runs are very similar to the graph shown in Figure 3.19 and were omitted due to space restrictions. Also the line for the multiple-packets case appears in the graph for comparison purposes, and will not be further explored in the rest of the section.

We performed a more extensive study to investigate the packet size requirements for networks and measuring sets of different sizes. The results are shown in Figures 3.20 and 3.21. In all figures, communication cost is measured by the number of transmissions, and packet size refers to the number of available slots (2 bytes each for our implementation) in each packet.

We observe that the required packet size appears to grow linearly with the network size, as well as the measuring set size, and a relatively small packet is sufficient to achieve a cost close to the optimal.

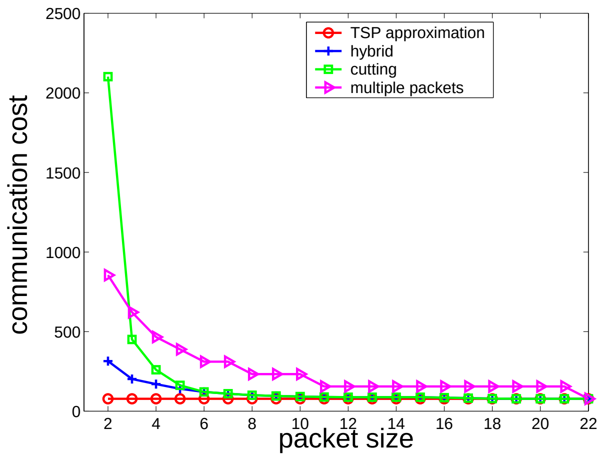


Figure 3.19. Communication cost of the 3 packet adjustment algorithms. This particular graph corresponds to a measuring set of size 15 in a network of 54 nodes.

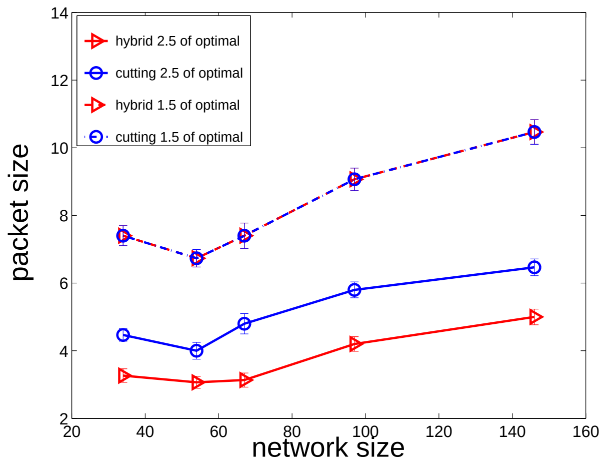


Figure 3.20. Packet size required for reaching a constant factor of the optimal cost, for networks of different size.

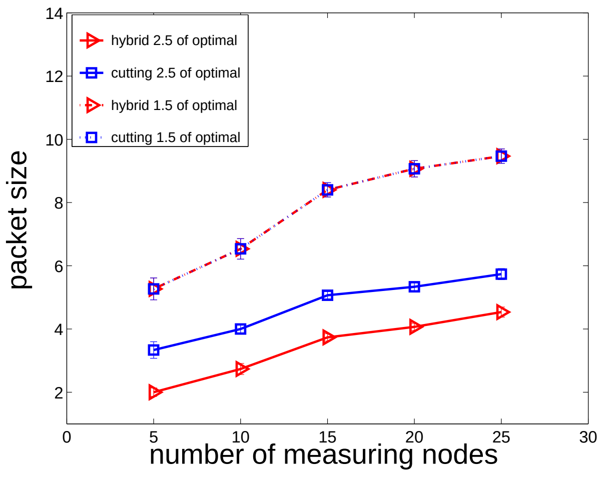


Figure 3.21. Packet size required for reaching a constant factor of the optimal cost for measuring sets of different size.

In terms of comparing the two main heuristics, cutting and hybrid, as expected hybrid demonstrates a lower communication cost. These results are verified by Figure 3.22 which was produced from experiments on the TOSSIM simulator. The figure compares the packet adjustment heuristics (hybrid and cutting) for two different distributions for picking the measuring set. One of them chooses uniformly from all the network nodes, and the other favors nodes that are positioned closer to the basestation.

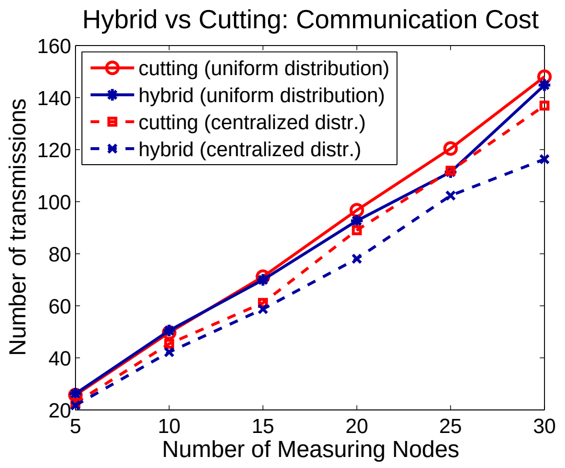


Figure 3.22. Comparison of the cost of the cutting and hybrid heuristics for measuring sets of various sizes chosen by two different distributions from all the network nodes.

These TOSSIM runs were performed by using packets of size 30 bytes. In our implementation the packet headers are of 8 bytes length, and each node slot requires 2 bytes. Therefore this corresponds to packets of size 11 (node slots being the measurement unit).

We also performed experiments for our proposed recovery approach. In addition to the previous setting, we pick a constant number of random failures in the network, and perform TOSSIM simulations for both our recovery algorithms. The depth used by the flooding recovery algorithm for the graphs that we present in this section is 3. This value was chosen after an evaluation that we did for various flooding depths, which we have omitted here due to space restrictions. For the network that we are modeling [13] in TOSSIM a bigger flooding depth didn't seem to add value to the recovery and even started to cause interference phenomena.

Figures 3.23 and 3.24 present experimental results for the two recovery approaches, corresponding to a 5%, a 10% and a 15% failure rate in the network. Figure 3.23 displays the overall communication cost for runs of various sizes for the measuring set. Each point in the graph is an average across 20 different runs of the same measuring set size. As the figure demonstrates, flooding is a more costly recovery technique compared to backtracking. Also, the backtracking cost is more robust to changes in the failure rate, since it is bounded by a constant factor of the cost of the original route, whereas such guarantees do not hold for flooding. In terms of the number of lost measurements, backtracking appears to win again, although the losses for both algorithms increase as the failure rate increases.

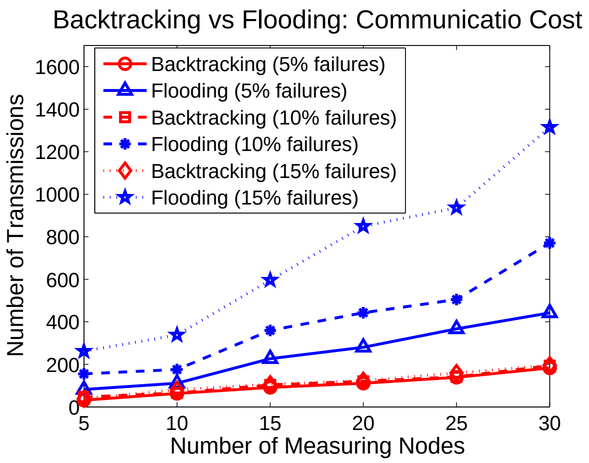


Figure 3.23. Comparison of the 2 recovery algorithms under conditions of failures with rates 5%, 10% and 15% in terms of communication cost. Notice that the backtracking lines practically coincide.

[13]The model is based on connectivity data gathered from the Mirage testbed.

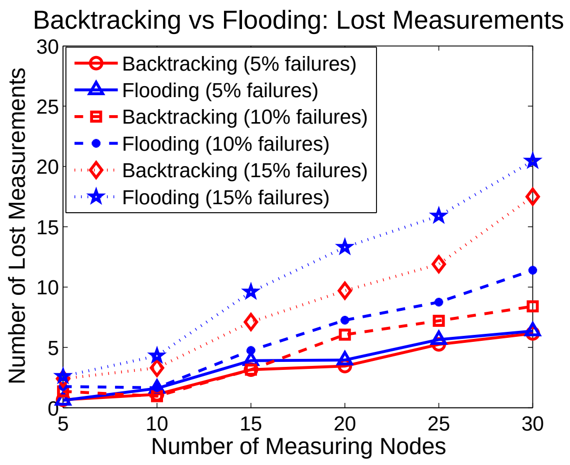


Figure 3.24. Comparison of the 2 recovery algorithms under conditions of failures with rates 5%, 10% and 15% in terms of the number of lost measurements.

3.7.2 Discussion

Our Matlab experiments demonstrate that the cutting and hybrid heuristics for adjusting long tours to finite packets converge to the optimal cost very fast, and for relatively small packet sizes. The TOSSIM experiments helped us evaluate our recovery schemes. Backtracking appears to be very robust to failures, with bounded cost and good recovery rates. We can always construct scenarios where backtracking loses to flooding, but in the network that we were simulating, it appears to be the winner. One of the main reasons was the poor quality of the communication links, which gave an advantage to backtracking which utilizes retries. This indicates that probably flooding would benefit from retransmissions (aggressive flooding) which could possibly include some acknowledgement scheme.

Chapter 4

Continuous Queries

Many practical applications require the monitoring of various physical phenomena that change over time. Sensing devices can often be hard to recharge, repair or replace, posing limitations to their use. These resource constraints demand nuanced schemes for collecting observations that tolerate bounded uncertainty in exchange for reduced resource consumption. In the previous chapter we proposed data gathering tours as a way to acquire measurements at minimum cost. Similar problems arise in other domains, for example, in robotic applications, there can be limits on fuel. Hence one has to plan robot trajectories in order to efficiently acquire information ([75]). Common to these problems is the need to find *maximally informative paths*, while *minimizing the traversal cost* incurred.

In the case of monitoring with sensor networks, the requirements for *informativeness* are usually determined by a user, who often specifies desired confidence requirements on the returned result. The workloads that we will study are similar to those of Chapter 3, where queries specify the desired informativeness based on an error window and confidence parameters, but in this Chapter we will extend the scope to continuous queries, which are repeated at given time intervals. In order to assess the informativeness of observations without acquiring the actual values, one can use probabilistic models. When dealing with spatial phenomena, as in the current common uses of sensor networks, *Gaussian Processes* [67] have been successfully used as such models. These models allow us to quantify the informativeness of selected observations in terms of the expected reduction in predictive variance.

Most existing work in the area of resource-bounded observation selection has been *myopic*, focusing optimization on a single timestep. In this chapter, we will present an efficient *nonmyopic* algorithm for observation selection in spatio-temporal models. Within its planning horizon, our algorithm optimizes a collection of paths, one for each time step, which minimizes the long-term observation cost. More specifically, the contributions are:

- An efficient nonmyopic observation planning algorithm with strong theoretical performance guarantees.
- A general technique, to convert any myopic planning algorithm into a nonmyopic algorithm.
- Empirical analyses of the effectiveness of our algorithm on real world data sets.

4.1 The Non-myopic Planning Problem

Our goal is to monitor a spatio-temporal phenomenon at a finite set of locations $\mathcal{V}$ and timesteps $\mathcal{T} = \{1, \ldots, T\}$. With each location i and time t, we associate a random variable $\mathcal{X}_{i,t}$, and use $\mathcal{V}_t$ and $\mathcal{X}_{\mathcal{V}_t}$ to refer to all locations and their variables at time t, respectively. Since it is costly to observe all locations at every point in time, our approach selects a set of locations to visit at each time step, and uses a statistical model to predict the missing values. To make this prediction, we assume a joint distribution $P(\mathcal{X})$ over all variables. This joint distribution encodes the dependencies along spatial and temporal dimensions. In this chapter, we use a class of nonparametric probabilistic models called Gaussian Processes (GPs, [67]), which have found successful use in modeling spatial phenomena. Our approach is however not limited to this class of models.

In order to select which observations to make, we need to quantify the expected "informativeness" of these observations with respect to the missing ones. We quantify the informativeness of any set of observations $\mathcal{A} = \cup_{1..t}\mathcal{A}_j$ by some function $f(\mathcal{A})$, where $\mathcal{A}_t$ refers to the observations made at time t. In the literature, different objective functions f have been proposed to quantify informativeness, such as entropy [73], mutual information [17], or the reduction of predictive variance [24]. In the case where no probabilistic model is available, one can also associate a sensing region observed by each sensor, and aim to maximize the total area covered [10]. All these objective functions have two properties in common. They are monotonic (i.e., $f(\mathcal{A}) \leq f(\mathcal{B})$ for $\mathcal{A} \subseteq \mathcal{B}$) and satisfy the following diminishing returns property: Adding a sensor to a small deployment helps more than adding it to a large one. More formally, $\forall \mathcal{A} \subseteq \mathcal{B} \subseteq \mathcal{V}$ and $s \in \mathcal{V} \setminus \mathcal{B}$, $f(\mathcal{A} \cup \{s\}) - f(\mathcal{A}) \geq f(\mathcal{B} \cup \{s\}) - f(\mathcal{B})$. A set function f satisfying this property is called *submodular* ([65]).[1] Here we will focus on optimizing a monotonic submodular set function f.

4.1.1 Submodularity and Informativeness

The submodularity property arises in many applications. Its applicability in our case can be seen through an example:

Assume a set of 5 nodes $\{n_1, n_2, \ldots, n_5\}$. Node n_1 is chosen in an observation set $A = \{n_1\}$. The observation of n_1 also contributes some information on the values of nodes n_3 and n_4. Currently, with $A = \{n_1\}$, the reward of adding n_3 to A is $R_{3:n_1}$. Say the observation of n_2 gives some information on n_3 and n_5, and suppose we augment A with n_2. Now, the reward of adding n_3 to the augmented observation set $A = \{n_1, n_2\}$ is $R_{3:n_1,n_2} < R_{3:n_1}$. Observing node 3 now becomes less useful, as the observation of node 2 "covered" some of the reward that n_3 would offer.

Clearly the benefit of observing each individual node decreases monotonically as the observation set gets larger, signifying that the information reward of each node satisfies the diminishing returns property. The benefit of adding element s to set A: $f(\mathcal{A} \cup \{s\}) - f(\mathcal{A})$ becomes smaller, as the size of A increases.

In our setting, we wish to solve a *filtering problem*: at each time t, we would like to predict the values of all unobserved variables in the current time step based on information collected up to this point in time, $\mathcal{A}_{1:t} = \cup_{1..t} \mathcal{A}_j$. In particular, given a measure of informativeness for each time step, f_t, we would like to ensure that the information collected up to that time step passes some user-specified threshold k_t, i.e., $f_t(\mathcal{A}_{1:t}) \geq k_t$. For example, a user may require that the average standard deviation of estimates for each temperature reading at each time step is smaller than $1^o C$.

Chosen observations at each time t must be connected into a path $\mathcal{P}_t = (v_0, \ldots, v_m)$ of nodes $v_i \in \mathcal{V}_t$. For all time steps, we assume we are given a nonnegative, possibly asymmetric distance function $d_t : \mathcal{V}_t \times \mathcal{V}_t \to \mathbb{R}^+$. In the wireless sensor network example, d_t

[1]Variance reduction is submodular under certain conditions ([24]). Mutual information is only approximately monotonic [38].

might reflect the expected transmission cost between any pair of locations at time t. The cost $C(\mathcal{P})$ of a path $\mathcal{P}$ can then be measured with respect to this distance function d_t.

We can now define our *nonmyopic spatiotemporal informative path planning problem (NSTIP).* Given a collection of submodular functions $f_t : 2^{\mathcal{V}_1 \cup \cdots \cup \mathcal{V}_t} \to \mathbb{R}^+$ defined on the timesteps up to t, cost functions d_t, and a set of accuracy constraints k_t we desire a collection of paths $\mathcal{P}_t$, one for each time step, with $\mathcal{P}^* = argmin_{\mathcal{P}} \sum_{t=1}^{T} C(\mathcal{P}_t)$ subject to $f_t(\mathcal{P}_{1:t}) \geq k_t \; \forall t$. Hereby, $\mathcal{P}_{1:t} = \cup_{1..t}\mathcal{P}_j$, where $\mathcal{P}_{t'} \subseteq \mathcal{V}_{t'}$ is the path selected at timestep t'. We will assume in discussion that the start and end nodes for each path are a single specified base-station node, but our algorithms extend to settings where we do not need to return to a base station at each time step.

4.2 Non-myopic Planning Algorithm

A *myopic* (greedy) approach for continuous querying is to treat each timestep as an independent single-step query, and optimize it independently. We are aiming for a *non-myopic* approach, that performs optimization by adjusting the rewards of observations to account for the effect that these can have for later timesteps.

In our approach to the NSTIP problem, we first convert the problem of optimizing multiple paths, one for each timestep, into a problem of optimizing a single path on a new graph, the *nonmyopic planning graph* (NPG). We then show how to use existing algorithms for solving a related problem, the *submodular orienteering problem* (SOP) [20] as a subroutine to solve our nonmyopic planning problem. This procedure will retain the approximation guarantees which existing SOP algorithms provide for the myopic case, while only introducing a small loss in the approximation guarantee due to the nonmyopic nature of our problem.

4.2.1 The Submodular Orienteering Problem

The Submodular Orienteering Problem is a variant of the Traveling Salesman Problem. The input to the general orienteering problem consists of an edge-weighted graph $G(V, E)$, a reward function f associated with every node in V, a source node s and a target node t, and a non-negative budget B. The goal is to find an "s-t" walk of total cost at most

B, so as to maximize the reward of the nodes participating in the path. In the case where the reward function f is *submodular*, then we have a case of the *Submodular Orienteering Problem* (SOP).

The hardness of SOP follows a simple reduction from the TSP, and therefore there is no polynomial algorithm for its solution. It is also known to be APX-hard to approximate [13]. The Chekuri-Pal algorithm [20] is a simple greedy algorithm that runs in quasi-polynomial time, that is shown to give an $O(\log OPT)$ approximation of the submodular orienteering solution.

4.2.2 The Nonmyopic Planning Graph

A solution for the NSTIP problem consists of a series of cyclic paths, one for each time step, starting and ending at the base-station node. Imagine these arranged on a timeline and connected through their base-station nodes as in Fig. 4.2.2(a).

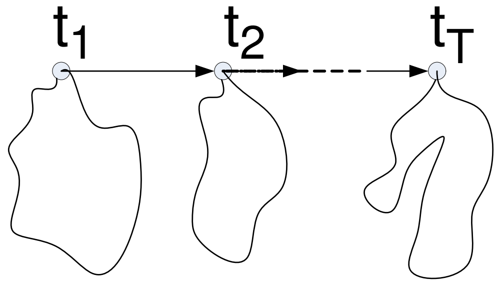


Figure 4.1. (a) Ex. NSTIP path.

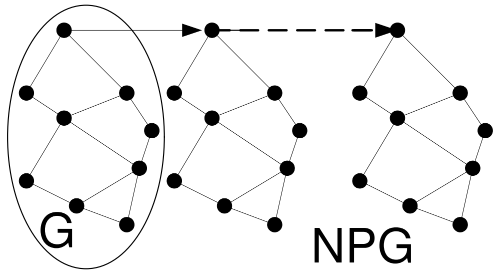


Figure 4.2. (b) Nonmyopic planning graph.

Fig. 4.2.2(a) represents the query plan across time. This overall solution, augmented by the edges connecting the basestation nodes at subsequent timesteps can now be considered a single path through a *different* graph. This *nonmyopic planning graph* (NPG) on *all* nodes $\mathcal{V}$ is constructed by combining the graph of finite-distance (via d_t) pairs of nodes from $\mathcal{V}_t$ for each timestep, adding zero-cost directed edges connecting them through the basestation nodes. This transformation is illustrated in Fig. 4.2.2(b).

Our goal is to recover a solution to the NSTIP problem by optimizing a path in the NPG. Note that any path $\mathcal{P}$ through the NPG, starting at the basestation at time 1, and ending there at time T, uniquely corresponds to a collection of paths $\mathcal{P}_t$, one for each timestep. Moreover, the cost of $\mathcal{P}$ is exactly the sum of the costs of all $\mathcal{P}_t$. We now need to define an objective function f and a constraint k on the NPG, such that a solution $\mathcal{P}$ on NPG is feasible (i.e., $f(\mathcal{P}) \geq k$) iff the corresponding collection of paths is feasible (i.e., $f_t(\mathcal{P}_{1:t}) \geq k_t$). To achieve this, we set $k = \sum_t k_t$, and for each timestep t redefine a function $f'_t(\mathcal{P}_{1:t}) = \min(f_t(\mathcal{P}_{1:t}), k_t)$. Hence, timestep t is satisfied iff $f'_t(\mathcal{P}_{1:t}) = k_t$. It can be verified that f'_t is still submodular and nondecreasing. Now define $f(\mathcal{P}) = \sum_t f'_t(\mathcal{P}_{1:t})$. f is nondecreasing and submodular, and $\mathcal{P}$ is a feasible solution iff $f(\mathcal{P}) = k$. Moreover, the set of optimal solutions coincides, having identical path costs.

4.2.3 Satisfying per-timestep constraints

Even after the described transformation, the problem of finding a minimum cost feasible path on the NPG is still NP-hard [30]. Nonetheless, since it is expensive to collect observations, we need an algorithm with theoretical guarantees to solve our problem. Unfortunately, we are unaware of any algorithm providing nontrivial guarantees for this problem. On the other hand, there are recent results by [20] on a basically dual problem – the *submodular orienteering problem* (SOP). Instead of minimizing cost for a fixed information threshold, SOP seeks a maximally-informative path $\mathcal{P}^*$ given a fixed budget B on path length. The cited paper presents a *recursive greedy* algorithm, which we call CP, which is guaranteed to return a path $\hat{\mathcal{P}}$ with cost at most B, such that $f(\hat{\mathcal{P}}) \geq \frac{1}{\alpha} f(\mathcal{P}^*)$, where $\alpha = \log |\mathcal{P}^*|$. Hence CP will return a solution where the reward is at most logarithmically worse in the length (number of nodes visited) of the optimal path. While the CP algorithm has the currently best known theoretical properties for SOP, our approach to NSTIP will accommodate *any* algorithm for SOP as a "black box". The approximation guarantee α will directly enter the approximation guarantees of our algorithm.

A naive approach would be to call an SOP solver with increasing budgets, until we satisfy the reward constraints. However, the SOP algorithm only gives an approximation guarantee for the reward of a particular budget, not on how much *more* budget is necessary to reach the desired reward. Instead, we will stop our search as soon as we satisfy some *portion* of the total achievable reward. To save time, we increase the budget values by powers of 2. Suppose that an optimal algorithm would fulfill the desired constraint ($f(P^*) = k$) with a path P^* with cost bounded by $2^j < C(P^*) \leq 2^{j+1}$, for some integer j. We then know that the approximate SOP algorithm, given a budget of $B = 2^{j+1}$ we will get a solution $\widehat{\mathcal{P}}_1$ with the guarantee $f(\widehat{\mathcal{P}}_1) \geq \frac{k}{\alpha}$.

However, since we covered only an α-fraction of the constraint k, we need to also cover the remaining difference. The key idea here is to look at the *residual reward*. For a set of "disqualified" nodes $\mathcal{A}$ (used in some earlier iteration), we define a new submodular function, $f_{\mathcal{A}}(\mathcal{B}) = f(\mathcal{A} \cup \mathcal{B}) - f(\mathcal{A})$,also monotonic. Our goal is to cover the missing portion of the constraint by optimizing this residual function. That is, if we have found a path $\widehat{\mathcal{P}}_1$ in the first iteration, we will fulfill our constraint with a path $\mathcal{P}$ such that $f_{\widehat{\mathcal{P}}_1}(\mathcal{P}) = k - f(\widehat{\mathcal{P}}_1)$. Since f is a monotonic function, we know that $f_{\widehat{\mathcal{P}}_1}(\mathcal{P}^*) = k - f(\widehat{\mathcal{P}}_1)$. Hence, there must exist a path (e.g., $\mathcal{P}^*$) of budget at most B which covers the remaining reward $k - f(\widehat{\mathcal{P}}_1) \leq k - \frac{k}{\alpha}$ when optimizing $f_{\widehat{\mathcal{P}}_1}$, i.e., when planning *conditionally* on the nodes we already observed.

Algorithm 3 $Cover(k)$

```
$Budget = 0$; $k_{cov} = 0$; $\mathcal{P} = \emptyset$
**while** $k_{cov} \leq k(1-\epsilon)$ **do**
  **for** $j = 1 \ldots j_{max}$ **do**
    $B = 2^j$; $\widehat{\mathcal{P}}' \leftarrow SOP_{\mathcal{P}}(B)$
    **if** $f(\widehat{\mathcal{P}}') \geq \frac{k - k_{cov}}{\alpha}$ **then**
      $Budget = Budget + B$; $k_{cov} = k_{cov} + f_{\mathcal{P}}(\widehat{\mathcal{P}})$
      $\mathcal{P} = \mathcal{P} \cup \widehat{\mathcal{P}}$; break
    **end if**
  **end for**
**end while**
```

By the above argument, we can again find an approximate solution $\widehat{\mathcal{P}}_2$ which covers an α fraction of the remaining difference. After m iterations, the remaining difference is $k - f(\widehat{\mathcal{P}}_1 \cup \cdots \cup \widehat{\mathcal{P}}_m) \leq (1 - \frac{1}{\alpha})^m k$, which shrinks exponentially fast in m. This process is described in Algorithm 3. Hereby, $SOP_{\mathcal{A}}$ for a set of nodes $\mathcal{A}$ denotes a call to the submodular orienteering blackbox, using the residual reward $f_{\mathcal{A}}$.

Since the reward function f is real valued, we need to specify a threshold ϵ, such that we are satisfied with a solution which covers a $(1-\epsilon)$ fraction of the desired accuracy constraint k. Theorem 6 bounds the number of iterations required in terms of the accuracy ϵ.

Theorem 6 *Algorithm 3 returns a solution of budget* $B \leq 2\frac{\log \epsilon}{\log(1-\frac{1}{\alpha})} B_{OPT}$ *which violates the constraint by at most* ϵk*, in time* $\mathcal{O}\left(\frac{\log \epsilon}{\log(1-\frac{1}{\alpha})} Q(nT, 2B_{OPT})\right)$.

Proof: At every iteration of the while loop of Algorithm 3, we cover an α-portion of the yet uncovered constraint, and the process will terminate when we have covered $k(1-\epsilon)$. At every step we are guaranteed not to exceed the budget of the previous step. This is because the optimal solution will always exist in the set of possible solutions that the SOP algorithm can pick. This optimal solution will have a reward that covers the constraints. Thus, based on the guarantees of the SOP algorithm, we know that for this budget the algorithm will return some set A' for which $f(A') \geq \frac{f(A_{OPT})}{\alpha}$. So, in every step, in order to cover an α-portion of the uncovered space, we will never need a budget bigger than 2^{j+1}, when the optimal budget is 2^j. This means that the SOP algorithm will never need to be called for a budget bigger than $2B_{OPT}$ if B_{OPT} is the budget of the optimal solution.

Also, at every step we aim for covering an α-portion of the uncovered constraint, so in iteration i the uncovered constraint would be $(1-\frac{1}{\alpha})^i k$. Since the algorithm will terminate when it has covered $\geq k(1-\epsilon)$, so the uncovered space would be $\leq k\epsilon$ (and thus the constraint cannot be violated by more than $k\epsilon$), we get that

$$(1-\frac{1}{\alpha})^i \leq \epsilon \Rightarrow i \leq \frac{\log \epsilon}{\log(1-\frac{1}{\alpha})}$$

So, we have a bound on the number of times that the while loop will be executed, which bounds the number of times that the SOP algorithm needs be called with the maximum budget ($2B_{OPT}$), in order to cover the reward constraints. If $Q(n, B)$ is the running time of the SOP blackbox for a graph of n nodes and for budget B, we know that we will call the SOP blackbox at most $\frac{\log \epsilon}{\log(1-\frac{1}{\alpha})}$ times on a graph of nT nodes and for budget $2B_{OPT}$.

Thus the running time of Algorithm 3 will be $O\left(\frac{\log \epsilon}{\log(1-\frac{1}{\alpha})} Q(n, B)\right)$

This also gives a bound on the total budget of the solution, since at every step we will never use more than $2B_{OPT}$ budget. So, our algorithm will give a solution with a budget no worse than $2\frac{\log \epsilon}{\log(1-\frac{1}{\alpha})} B_{OPT}$.

■

Hereby, $Q(n, B)$ is the running time of the SOP black box executed on a graph with n nodes and budget B. For the CP algorithm, $Q(n, B) = \mathcal{O}((nB)^{\log(n)})$.

4.3 Efficient Non-myopic Planning

In the previous section we presented a transformation of a multistep problem to a single step equivalent, which allows for the use of existing algorithms with good theoretical bounds. Additionally we demonstrated how an approximation algorithm of a dual problem, the Submodular Orienteering Problem, can be adapted to address our setting, while preserving the algorithm's guarantees.

This approach has running time proportional to $Q(nT, B)$, i.e. the running time of the submodular orienteering blackbox executed on a graph with nT nodes, and takes advantage of the algorithm's guarantee. Currently the best known guarantee is the one by [20]. Unfortunately, the guarantee of their approach comes at a price – albeit subexponential, the algorithm is superpolynomial: For our setting, their running time is bounded by $\mathcal{O}((nTB)^{\log(nT)})$. Using a spatial decomposition approach and branch and bound techniques, this running time can empirically be significantly decreased [75]. However, the Nonmyopic Planning Graph (NPG) gets very big very quickly, even for small horizons T, quickly rendering the approach infeasible.

4.3.1 Nonmyopic Greedy Algorithm

In this section we will present an efficient greedy algorithm, which can be used more effectively in large structures like the Nonmyopic Planning Graph, and which also provides a theoretical bound better than that of [20] on the NPG. What makes the development of an efficient algorithm possible in this scenario is the flexibility of Algorithm 3, which was developed based on a blackbox understanding of the SOP Algorithm. Algorithm 3 simply uses a solution to the dual SOP problem, and transforms it to a solution of the NSTIP problem. What makes the algorithm computationally expensive is the use of the SOP algorithm on the NPG graph which is big in size. To develop an efficient alternative we will replace the blackbox that solves the SOP on NPG with another algorithm which is based on a greedy selection on every step, and resorts to the SOP algorithm on the smaller network graph.

In the same spirit as in our previous analysis, the reward that we gain from observing a set $\mathcal{A}$ is given by $f(\mathcal{A})$ which is a submodular monotone function. The marginal increase of f with respect to $\mathcal{A}$ and X is defined as $f'(\mathcal{A}; X) = f(\mathcal{A} \cup X) - f(\mathcal{A})$. All elements are chosen from a set $\mathcal{V}$, and for a positive budget B, our greedy blackbox will solve the budgeted maximization problem:

$$OPT = argmax_{\mathcal{A}\in\mathcal{V}:c(\mathcal{A})\leq B} f(\mathcal{A})$$

An element X in the greedy algorithm setting corresponds to a tour in the network graph at a particular timestep, and the approximately best tours are computed for each assignment of possible budgets per timestep. The SOP algorithm is used as an approximation oracle that returns a tour X with an α approximation factor relative to the optimal solution.

The details of our greedy algorithm are given in Algorithm 4.

Algorithm 4 *NonmyopicGreedy*$(k, Budget, T)$

```
$\mathcal{A}_1 = \emptyset$
$usedB = 0$
while $usedB \leq Budget$ do
  for $t = 1 \ldots T$ do
    for $b = 1 \ldots Budget - usedB$ do
      $\mathcal{M}(b, T) = SOP(b, \mathcal{G}_t, \mathcal{A}_1)$
    end for
  end for
  $X^* = argmax\{f'(\{\mathcal{A}_1; X_{b,t}\})/c(X_{b,t}) : X_{b,t} \in \mathcal{M}\}$
  $usedB = usedB + c(X^*)$
  $\mathcal{A}_1 = \mathcal{A}_1 \cup X^*$
end while
% Given $\mathcal{A}_1$ compute the best greedy choice for $Budget$
for $t = 1 \ldots T$ do
  $\mathcal{M}_2(Budget, t) = SOP(Budget, \mathcal{G}_t, \mathcal{A}_1)$
end for
$\mathcal{A}_2 = argmax\{f'(\{X_{b,t}\})/c(X_{b,t}) : X_{b,t} \in \mathcal{M}_2\}$
return $argmax_{\mathcal{A}\in\{\mathcal{A}_1,\mathcal{A}_2\}} f(\mathcal{A})$
```

The algorithm begins by filling in a $B \times T$ matrix, where element (b, t) contains the orienteering solution over the network graph that corresponds to timestep t and for budget b. The greedy rule adds to set $\mathcal{A}$ the element X^* such that

$$X^* = \max_{X\in\mathcal{W}\setminus\mathcal{G}_{i-1}} \frac{f'(\mathcal{G}_{i-1}; X)}{c(X_i)},$$

where $\mathcal{W}$ is the set of all possible choices for X and $\mathcal{G}_{i-1}$ the already chosen set. Since we can't evaluate the marginal increases $f'(\mathcal{G}_{i-1}; X)$ exactly, we only assume that we can evaluate $\hat{f}(\mathcal{G}_{i-1}; X)$, using the SOP algorithm as an approximation oracle. The main part of the algorithm is the computation of the greedy set of tours $\mathcal{A}_1$. However there are a few cases when $\mathcal{A}_1$ can be arbitrarily bad. These are covered by $\mathcal{A}_2$, which is the best tour when assigning all of the given budget to a single timestep, given set $\mathcal{A}_1$. $\mathcal{A}_2$ is guaranteed to be good in these few corner cases. Despite its greedy nature, our algorithm is still nonmyopic, as the reward functions consider the effect of node selections in future timesteps as well.

Algorithm 4 provides a solution to the SOP over multiple timesteps, which is equivalent to a solution on the NPG graph, and thus can replace the blackbox call to the SOP in algorithm 3. This is computationally an improvement, as the SOP procedure is now called on the smaller network graph as opposed to the NPG. In Algorithm 4 the initial population of the matrix requires $B \times T$ calls to the SOP, and the while loop will repeat this process at most B times. Thus, the running time of Algorithm 4 is $\mathcal{O}(B^2 T Q(n, B))$, where $Q(n, B)$ is the running time of the SOP executed on a graph of n nodes and budget B. Specifically, if the CP algorithm is used for the SOP calls, the final running time will be $\mathcal{O}(B^2 T (nB)^{\log n})$, as opposed to $\mathcal{O}((nTB)^{\log(nT)})$ that the call to the SOP on the NPG graph would cost.

Additionally to improving the running time, Algorithm 4, also provides better approximation bounds than the SOP on the NPG offers, which for the CP algorithm is $\frac{1}{\log(nT)}$. This statement is formally given by Theorem 7:

Theorem 7 *Algorithm 4 returns a solution path P for which*

$$f(P) \geq \frac{1}{2}\left(1 - \frac{1}{e^{\frac{1}{\alpha}}}\right) f(OPT) - \frac{1}{2}\beta\varepsilon$$

where α is the approximation factor of the SOP call.

Proof: We will consider the computation of the set $\mathcal{A}_2$ in Algorithm 4. We name $\mathcal{V}$ all the possible choices of X, i.e. every element of the matrix M. We renumber $\mathcal{V} = \{X_1, \dots, X_n\}$ and define $\mathcal{G}_0 = \emptyset$ and $\mathcal{G}_i = \{X_1, \dots, X_i\}$ such that

$$\frac{f(\mathcal{G}_i) - f(\mathcal{G}_{i-1})}{c(X_i)} \geq \max_Y \frac{f(\mathcal{G}_{i-1} \cup Y) - f(\mathcal{G})}{\alpha c(Y)}$$

for some $\alpha \geq 1$, which is the approximation factor of the approximation oracle. The sequence $(G_j)_j$ corresponds to the sequence of assignments to $\mathcal{A}_2$, and is motivated by the simple greedy rule, adding, for a prior selection $\mathcal{G}_{i-1}$, the element X_i such that

$$X_i = \max_{X \in \mathcal{W} \setminus \mathcal{G}_{i-1}} \frac{\hat{f}(\mathcal{G}_{i-1}; X)}{c(X_i)}.$$

Let $l = \max\{i : c(\mathcal{G}_i) \leq B\}$ be the index corresponding to the iteration, where $\mathcal{A}_2$ is last augmented, hence $\mathcal{A}_2 = \mathcal{G}_l$. Let $L = c(OPT)$, $c_{min} = \min_X c(X)$ and $w = |OPT|$.

To prove Theorem 7, we need two lemmas:

Lemma 3 *For $i = 1, \ldots, l+1$, it holds that*

$$f(\mathcal{G}_i) - f(\mathcal{G}_{i-1}) \geq \frac{c(X_i)}{\alpha L}(f(OPT) - f(\mathcal{G}_{i-1})) - \varepsilon \left(1 + \frac{wc(X_i)}{\alpha L}\right)$$

Proof: Using monotonicity of f, we have

$$f(OPT) - f(\mathcal{G}_{i-1}) \leq f(OPT \cup \mathcal{G}_{i-1}) - f(\mathcal{G}_{i-1}) = f(OPT \setminus \mathcal{G}_{i-1} \cup \mathcal{G}_{i-1}) - f(\mathcal{G}_{i-1})$$

Assume $OPT \setminus \mathcal{G}_{i-1} = \{Y_1, \ldots, Y_m\}$, and let for $j = 1, \ldots, m$

$$Z_j = f(\mathcal{G}_{i-1} \cup \{Y_1, \ldots, Y_j\}) - f(\mathcal{G}_{i-1} \cup \{Y_1, \ldots, Y_{j-1}\}).$$

Then $f(OPT) - f(\mathcal{G}_{i-1}) \leq \sum_{j=1}^{m} Z_j$.

Now notice that

$$\frac{Z_j - \varepsilon}{\alpha c(Y_j)} \leq \frac{f(\mathcal{G}_{i-1} \cup Y_j) - f(\mathcal{G}_{i-1}) - \varepsilon}{\alpha c(Y_j)} \leq \frac{f(\mathcal{G}_i) - f(\mathcal{G}_{i-1}) + \varepsilon}{c(X_i)}$$

using submodularity in the first and the greedy rule in the second inequality. Since $\sum_{j=1}^{m} c(Y_j) \leq L$ it holds that

$$f(OPT) - f(\mathcal{G}_{i-1}) = \sum_{j=1}^{m} Z_j \leq \alpha L \frac{f(\mathcal{G}_i) - f(\mathcal{G}_{i-1}) + \varepsilon}{c(X_i)} + m\varepsilon$$

■

Lemma 4 *For $i = 1, \ldots, l+1$ it holds that*

$$f(\mathcal{G}_i) \geq \left[1 - \prod_{k=1}^{i}\left(1 - \frac{c(X_k)}{\alpha L}\right)\right] f(OPT) - \left(\frac{\alpha L}{c(X_i)} + w\right)\varepsilon.$$

Proof: Let $i = 1$ for sake of induction. We need to prove that $f(\mathcal{G}_1) \geq \frac{c(X_1)}{\alpha L} f(OPT) - \left(\frac{\alpha L}{c(X_i)} + w\right)\varepsilon$. This follows from Lemma 3 and since

$$\frac{\alpha L}{c(X_i)} + w \geq 1 + \frac{wc(X_i)}{\alpha L}.$$

Now let $i > 1$. We have

$$\begin{aligned}
f(\mathcal{G}_i) &= f(\mathcal{G}_{i-1}) + [f(\mathcal{G}_i) - f(\mathcal{G}_{i-1})] \\
&\geq f(\mathcal{G}_{i-1}) + \frac{c(X_i)}{\alpha L}[f(OPT) - f(\mathcal{G}_{i-1})] - \varepsilon\left(1 + \frac{wc(X_i)}{\alpha L}\right) \\
&= \left(1 - \frac{c(X_i)}{\alpha L}\right) f(\mathcal{G}_{i-1}) + \frac{c(X_i)}{\alpha L} f(OPT) - \varepsilon\left(1 + \frac{wc(X_i)}{\alpha L}\right) \\
&\geq \left(1 - \frac{c(X_i)}{\alpha L}\right)\left[\left(1 - \prod_{k=1}^{i-1}\left(1 - \frac{c(X_k)}{\alpha L}\right)\right) f(OPT) - \left(\frac{\alpha L}{c(X_i)} + w\right)\varepsilon\right] + \\
&\quad + \frac{c(X_i)}{\alpha L} f(OPT) - \varepsilon\left(1 + \frac{wc(X_i)}{\alpha L}\right) \\
&= \left(1 - \prod_{k=1}^{i}\left(1 - \frac{c(X_k)}{\alpha L}\right)\right) f(OPT) - \varepsilon\left(1 + \frac{wc(X_i)}{\alpha L}\right) - \left(\frac{\alpha L}{c(X_i)} + w\right)\varepsilon\left(1 - \frac{c(X_i)}{\alpha L}\right) \\
&= \left(1 - \prod_{k=1}^{i}\left(1 - \frac{c(X_k)}{\alpha L}\right)\right) f(OPT) - \left(\frac{\alpha L}{c(X_i)} + w\right)\varepsilon
\end{aligned}$$

using Lemma 3 in the first and the induction hypothesis in the second inequality. ■

From now on let $\beta = \frac{\alpha L}{c_{min}} + w$.

Observe that for $a_1, \ldots, a_n \in \mathbb{R}^+$ such that $\sum a_i = A$, the function $(1 - \prod_{i=1}^{n}(1 - \frac{a_i}{\alpha A}))$ achieves its minimum at $a_1 = \cdots = a_n = \frac{A}{n}$. In order to show this, let $G(a_1, \ldots, a_{m-1}) = \log \frac{A - \sum_{j=1}^{m-1} a_j}{A} + \sum_i \log(\alpha - \frac{a_i}{A})$. It holds that

$$\frac{\partial G}{\partial a_i} = \frac{-1/A}{\alpha - a_i/A} + \frac{1/A}{\alpha - (A - \sum_{j=1}^{m-1} a_j)/A}.$$

This derivative is 0 iff $A - \sum_{j=1}^{m-1} a_j = a_i$ for all $1 \leq i < m$. Hence, for $1 \leq i \leq m$ it must hold that $a_i = A/m$. We hence have

$$\begin{aligned}
f(\mathcal{G}_{l+1}) &\geq \left[1 - \prod_{k=1}^{l+1}\left(1 - \frac{c(X_k)}{\alpha L}\right)\right] f(OPT) - \beta\varepsilon \\
&\geq \left[1 - \prod_{k=1}^{i}\left(1 - \frac{c(X_k)}{\alpha c(\mathcal{G}_{l+1})}\right)\right] f(OPT) - \beta\varepsilon \\
&\geq \left[1 - \left(1 - \frac{1}{\alpha(l+1)}\right)^{l+1}\right] f(OPT) - \beta\varepsilon
\end{aligned}$$

$$\geq \left(1 - \frac{1}{e^{1/\alpha}}\right) f(OPT) - \beta\varepsilon$$

where the first inequality follows from Lemma 4 and the second inequality follows from the fact that $c(\mathcal{G}_{l+1}) > L$, since it violates the budget.

Name X_{l+1} the second candidate solution considered by the algorithm. From submodularity and monotonicity we get:

$$f(\mathcal{G}_l) + f(X_{l+1}) \geq f(\mathcal{G}_{l+1}) \geq \left(1 - \frac{1}{e^{1/\alpha}}\right) f(OPT) - \beta\varepsilon$$

thus

$$\max\{f(\mathcal{G}_l), f(X_{l+1})\} \geq \frac{1}{2}\left(1 - \frac{1}{e^{1/\alpha}}\right) f(OPT) - \beta\varepsilon$$

■

Specifically, for $\alpha = \log n$, the bound becomes as follows:

Since n represents the nodes in the network, n is an integer ≥ 2. Now, $\forall x \in [0,1]$ and $\forall \eta$, it is $e^{\eta x} \leq 1 + (e^{\eta} - 1)x$. Replacing x with $\frac{1}{\log n}$ and $\eta = -1$ we get:

$$e^{-\frac{1}{\log n}} \leq 1 + (e^{-1} - 1)x \Rightarrow 1 - \frac{1}{e^{\frac{1}{\log n}}} \geq (1 - e^{-1})\frac{1}{\log n}$$

Using this in the previous bound we get:

$$f(P) \geq \frac{1 - e^{-1}}{2 \log n} f(OPT)$$

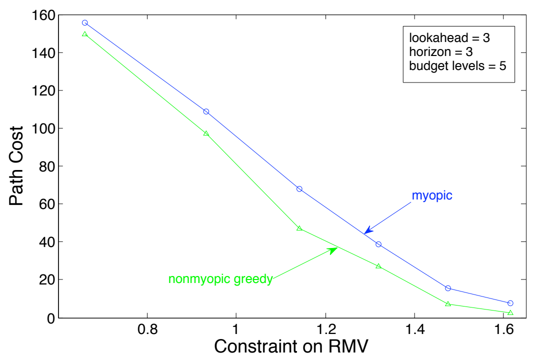
More specifically, if the approximation factor of the SOP blackbox is $\log n$ as is the case with the CP algorithm, it is easy to show that $f(P) \geq \frac{1-e^{-1}}{2\log n} f(OPT)$.

4.3.2 Adaptive Discretization

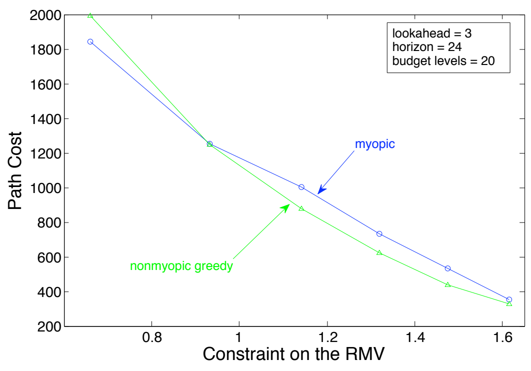
To compute an entry $M(b,t)$ in Algorithm 4, we must perform an expensive call to the SOP blackbox for each value of $b \leq B$. This computational cost can quickly become prohibitive if we make the computation for all budget levels. To counter that we need a discretization scheme, and only make the expensive SOP call for some relatively small number of budget levels. Instead of choosing an arbitrary discretization approach, we propose an *adaptive discretization* which, although heuristic in nature, was found to perform well empirically; a small number of budget levels (roughly $O(T)$) sufficed to achieve good solutions.

Algorithm 5 *Adaptive*(k, B, T)

```
for $t = 1$ to $T$ do
  $\tilde{\mathcal{B}}_t = \{b_0 = 1, b_1 = B\}\}$
  $rew[b_0]$=compRew$(b_0)$; $rew[b_1]$ = compRew$(b_1)$
  for $j = 2$ to MaxNumLevels do
    $i = \arg\max_i (rew[b_{i+1}] - rew[b_i]) \cdot (b_{i+1} - b_i)$
    $b' = (b_{i+1} + b_i)/2$
    $rew[b']$ = compRew$(b')$ using SOP blackbox
    insert $b'$ into $\tilde{\mathcal{B}}_t$; store reward of b'
  end for
end for
```



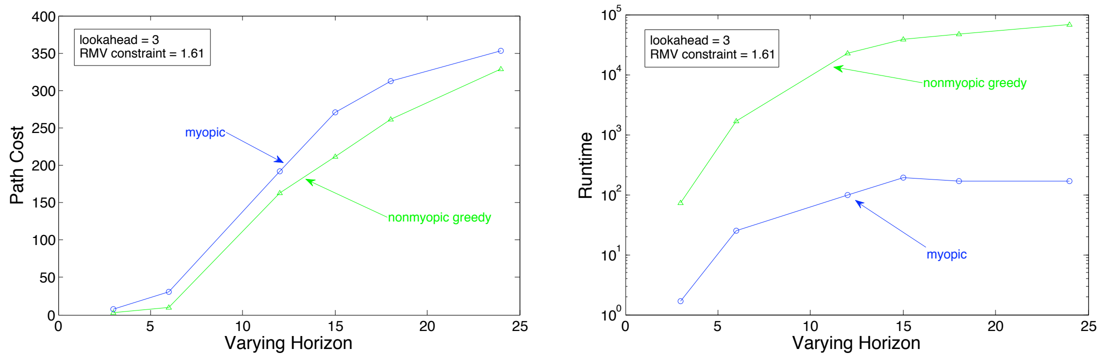
(a) Varying constraints: T=3, LA=3



(b) Varying constraints: T=24, LA=3

Figure 4.3. Algorithm comparison for varying constraints

We dynamically decide which entries $M(b, t)$ to compute, as shown in Alg. 5. For each timestep t we maintain a sorted list of budget levels $\tilde{\mathcal{B}}_t$, initially containing only 1 and B, and the rewards of all budget levels. We iteratively add a new level between b_i and b_{i+1}, such that $(rew(i+1) - rew(i)) \cdot (b_{i+1} - b_i)$ is maximized. The reason is that we want to add discretization where the reward difference is large, but also where the gap between budgets is large.



(a) Varying horizon: RMS≤1.61, LA=3 (b) Runtime comparison(d)

Figure 4.4. Algorithm comparison for varying horizon

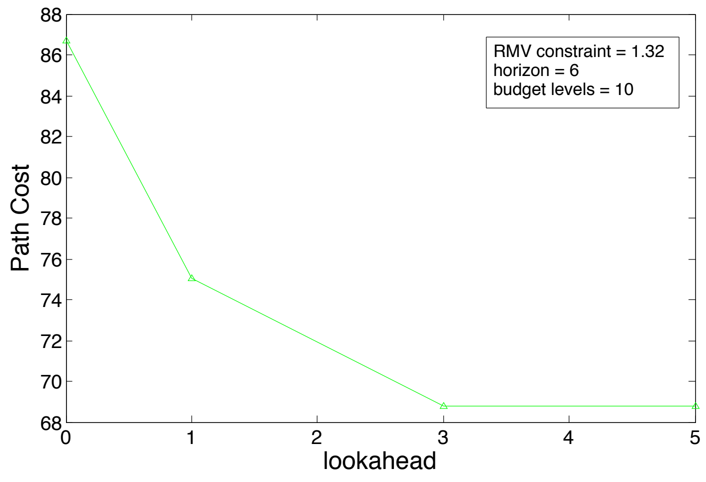
4.4 Evaluation

We empirically analyze our proposed algorithm, comparing the myopic and nonmyopic greedy approaches for various settings of our parameters. We also demonstrate how these parameters affect the running time and the quality of the results.

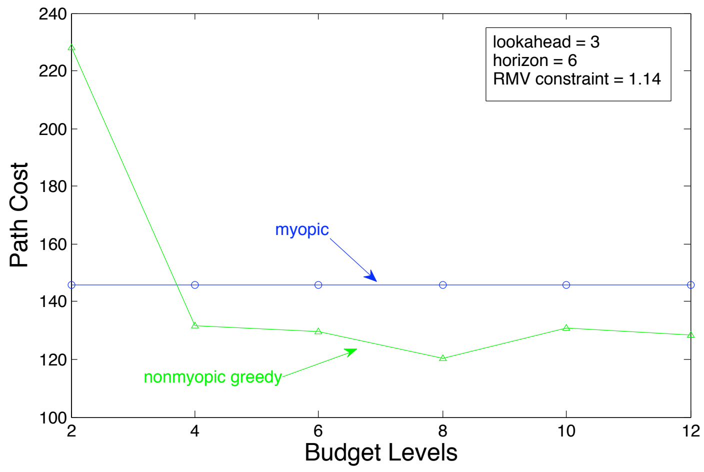
Our experiments are based on a real data set. Data was gathered from a deployment of a 46 node sensor network at the Intel Berkeley Lab used by [28]. Temperature and network connectivity measurements were gathered at one hour intervals, for a period of seven days. We learned Kalman filter models, capturing the satiotemporal correlations. Five days of measurements were used for training and learning the transition model, and two were used for testing.

4.4.1 Implementation Details

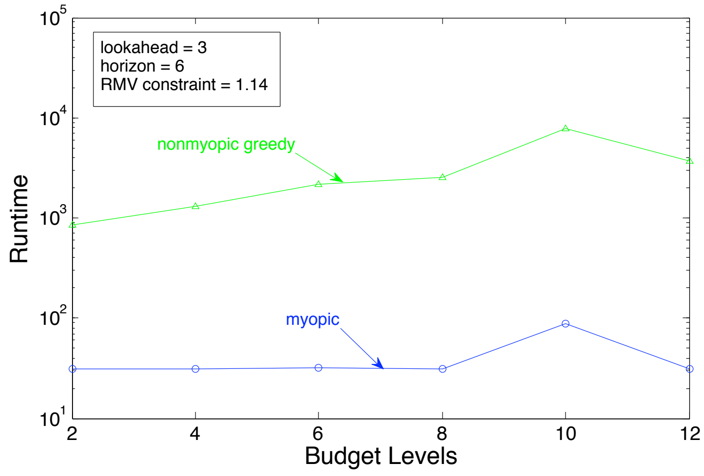
In our experiments we compare the myopic to the non-myopic greedy approach. The myopic approach calls Alg. 3 once for each timestep, the objective function only considers the current timestep and the model is updated based on the chosen observations before proceeding to the next timestep. The nonmyopic algorithm uses the greedy approach with adaptive discretization to produce an observation plan for multiple timesteps. The objective function considers the reward that a set has from the current to a fixed number of timesteps into the future: $f_t'(\mathcal{A}_{1:t}) = \sum_{t'=t}^{\min(t+LA,T)} f_{t'}(\mathcal{A}_{1:t})$. T is the planning horizon, LA a lookahead



(a) Varying lookahead: T=6, RMS≤1.32



(b) Varying discretization: T=6, RMS≤1.14



(c) Runtime for different budget levels (f)

Figure 4.5. Varying the parameters of the nonmyopic greedy algorithm

parameter that allows us to interpolate between the myopic solution ($LA = 1$), to the completely nonmyopic approach ($LA = T$).

Both algorithms use a fast heuristic by [18] as blackbox for the SOP problem, which [75] experimentally showed to often provide relatively good results when compared to their efficient version of the recursive greedy algorithm of [20].

In many practical applications, one wants to control the RMS error. Hence, in our implementation we chose the mean total variance reduction as our objective function: $f_t(\mathcal{A}_{1:t}) = \frac{1}{n}\sum_i (Var(\mathcal{X}_{i,t}) - Var(\mathcal{X}_{i,t} \mid \mathcal{X}_{\mathcal{A}_{1:t}}))$, which is monotonically nondecreasing, but not always submodular. There is both empirical and theoretical evidence however that in many practical problems, it indeed is submodular [24]. So, for our per-timestep information constraints, we chose the Root Mean Variance (RMV), since this is the criterion used in the myopic approach of [28], requiring $\sqrt{\frac{1}{n}\sum_i Var(\mathcal{X}_{i,t} \mid \mathcal{X}_{\mathcal{A}_{1:t}})} \leq k$, where k is the maximum RMV allowed each timestep.

4.4.2 Experiments

In our first set of experiments, we test how the accuracy constraint k affects the path cost achieved by both algorithms. Figures 4.3(a) and 4.3(b) display how the query path costs of both algorithms change as we vary the constraints on the RMV. In both graphs, we used a lookahead of 3 steps in the objective function. The first graph presents plans for a horizon of 3 hours, whereas the second uses a horizon of 24 hours. The path costs decrease as we loosen the constraints on the RMV. Apart from one data point, our nonmyopic algorithm always dominates the myopic one, providing better solutions. This outlier is due to the adaptive discretization procedure. For rather loose constraints, we observe that the nonmyopic algorithm drastically outperforms the myopic algorithm, often providing a reduction in cost of up to 30%. For very loose constraints, as well as for very tight constraints, the performance of both algorithms is very similar. In the "loose" case, few observations close to the basestation suffice, a solution found by both algorithms. In the "tight" case, both algorithms choose a large number of observations to satisfy the constraints, so the nonmyopic benefit is decreased.

In our second set of experiments (Fig. 4.4(a)) we observe how the overall path cost of the two algorithms varies with the number of planning timesteps of the query. The RMV constraint used for this experiment was 1.6 and the lookahead of the non-myopic greedy algorithm was set to 3. We observe that our non-myopic approach is consistently better. Relative to the number of timesteps, observe that the biggest improvement happens between 0 and 6 timesteps, when the nonmyopic algorithm drastically outperforms the myopic algorithm. This gain can be explained by noting that the experiment starts around midnight. In the first few hours, the model is very accurate in predicting the temperatures from very few observations. During daytime hours, the path cost quickly increases as more observations have to be made. In the late evening to night time, again, few observations suffice for accurate predictions, and the nonmyopic algorithm again outperforms the myopic approach. In Fig. 4.4(b) we see how the running time of the greedy algorithm is affected by the number of timesteps we need to plan for. As expected from our analysis of the running times, they grow linearly in terms of the horizon.

In our third set of experiments we examine how the various parameters of our greedy algorithm affect its performance. In Fig. 4.5(a) we examine how the lookahead parameter of the objective function affects the quality of the result for the nonmyopic approach. The experiment is performed for a planning horizon of 6 timesteps, and for a RMV constraint of 1.3. We see that, as the lookahead increases, the results get better, which is evidence that nonmyopic planning is very important for continuous queries. The biggest improvement is achieved from the myopic setting for the objective function (lookahead equal to 0) to a lookahead of 1.

In the following two graphs, we examine how the adaptive discretization in the greedy algorithm affects its performance. Figures 4.5(b) and 4.5(c) display the results of this experiment for a horizon of six timesteps and a constraint on the RMS of 1.14. The first figure displays how the cost of the query plan produced by the non-myopic algorithm changes with the number of discretization levels. For very coarse discretization (few budget levels), the non-myopic solution is worse than the myopic result, because the coarse discretization prunes too much of the nonmyopic algorithm's search space, and good solutions are lost. Increasing the number of budget levels to roughly the order of timesteps, the solutions improve and exceed the performance of the myopic algorithm. As expected, finer discretization is more desirable, but as Fig. 4.5(c) shows, the running time increases.

4.5 Discussion of Related Work

The model-driven scheme proposed by the BBQ system [28] is myopic. Using either an exponential or greedy algorithm, observation plans are built only for the current timestep, without any provisioning for continuous queries.

[54] consider the problem of nonmyopic collaborative target tracking in sensor networks. They nonmyopically optimize the information obtained from a sequence of observations. To optimize this criterion, they perform a heuristic “min-hop” search without approximation guarantees. Empirically their results shed more evidence on the importance of nonmyopic sensor selection.

Our problem is also related to the *Traveling Salesman Problem with profits* (TSPP; [31]). In TSPP, each node has a fixed reward and the goal is to find a path that maximizes the sum of the rewards, while minimizing the cost of visited nodes. The orienteering problem is a special case of TSPP, maximizing rewards, subject to constraints on the cost [49]. There are several important differences to this body of work. The TSPP objective is a *modular* function. When selecting informative observations however, closeby locations are correlated, and hence their information is sub-additive (submodular). Furthermore, our approach is nonmyopic, planning multiple paths, satisfying constraints for each time step. Our algorithm is efficient with respect to the planning horizon T, and provides approximation guarantees.

In robotics, similar work was developed in the context of *simultaneous localization and mapping* (SLAM). [78] develop a greedy algorithm, without approximation guarantees, for selecting the next location to visit to maximize information gain about the map. [74] attempt to optimize the entire trajectory, not just the next step, but their algorithm introduces some approximation steps without theoretical bounds. We also expect our approach to be useful in the SLAM setting.

Chapter 5

Distributed Modeling

In the previous chapteres we focused on computing optimal plans for certain data gathering problems. Planning was based on centralized knowledge of the problem parameters, which was used to design communication solutions that minimize the overall communication cost. Such centralized approaches are as good as the accuracy of the models and the parameters upon which they are based. Unexpected events like node failures, changes in connectivity and model inaccuracies can reduce the effectiveness of the given solutions. Also, given the highly distributed nature of sensor networks, more localized models are easier to maintain.

In this chapter we maintain a model-based approach to approximating queries, but we move implementation to an in-network setting, which provides a number of features: models can be kept up-to-date more easily, pre-processing of routing paths at query runtime can be eliminated, and failures on routing paths are no longer crucial. We propose the use of a carefully designed in-network spanning tree to minimize the communication required to return a robust estimate of the value reported by each sensor node. This tree is rooted at a base station, with Gaussian models stored at each node in the tree, one per child of the node. Queries are answered by traversing to a depth in the tree where the summaries provide sufficient information to answer a query within a specified window of accuracy.

5.1 Related Work

The idea of Semantic Routing Trees was proposed in [57] as an overlay index in the network, to allow for routing decisions to those leaves relevant to the query. Our in-network summaries go further, maintaining summaries of the data at different tree levels to allow for reduction in communication. The use of summaries for query processing has been examined in different settings. GHTs [68] propose storing and retrieving network information using Geometric Hash Tables, and Distributed Quadtrees [26] overlay quadtree structures over WSNs to satisfy distance sensitive spatial queries.

Cristescu et al ([22], [23]) study the relationship between data representation and data gathering, based on coding strategies. For a gathering task spanning all network locations, the goal is to minimize communication by optimizing the tree routing structure for a given coding model. The problem of jointly optimizing sensor placement and the transmission structure is further developed in [32].

Our in-network summaries use models distributed in the network to make query processing more cost efficient. The same objective was tackled by centralized approaches: In [28], the BBQ system proposes a model-driven scheme to provide approximate answers to queries posed in a sensor network, satisfying some information guarantees. [63], [64] also focus on a centralized approach, where all the decisions and planning are performed at a basestation node, and heuristics are given to cope with unexpected network behavior. PRESTO [51] is an in-network model-driven scheme, which is however push based – deviations for the model are detected locally and in this case data is pushed to the root.

The TinyDB system [59], which is largely used for data collection in sensor networks, uses spanning trees for the data retrieval, but does not rely on any other in-network data to optimize queries. The role of in-network storage is discussed in [61] as a way to reduce energy consumption in its role in the trade-off between computation and communication. Maintaining data in the network is the focus of distributed storage ([29]). Several different coding schemes are developed for storing information in the network, but the goal in this case is to make the data resilient to failures and not to optimize query execution. A similar tactic is employed by the SPIN protocol [44], [48], which disseminates data in the network, so that a user posing a query at different locations can immediately get back results.

In terms of data gathering, directed diffusion [46] sets up gradients from data sources to the basestation, forming paths of information flow, which also perform aggregation. Rumor Routing [16], uses long lived agents that create and redirect paths to events they encounter.

We break this problem into individual tasks which we discuss in the corresponding sections. More specifically, this chapter contributes the following:

- A definition of the problem of optimal in-network summaries.
- An efficient Gaussian-based compression scheme that is geared towards minimizing erroneous reporting of values, and can be optimized based on query workload.
- Query traversal algorithms that utilize the compression scheme to make routing decisions and give value estimates with limited communication.
- Hardness results for the optimal in-network summary problem, along with approximation algorithms and heuristics that we evaluate experimentally.
- Experimental evaluation of the sensitivity of our approach to changes in the data and query workload.

5.2 In-network Summaries

In this section we focus on the type of data summaries that we need to maintain to optimize queries in a sensornet setting and give a definition of optimality for that scheme. We then define the problem of data compression and treat it from the standpoint of the query workload.

Our workload consists of "`SELECT *`" style queries that request the approximate values of multiple sensors, without aggregation. As was the case in the previous chapters, the approximation bounds are defined by an accuracy window w and confidence limit δ specified by a query Q ($Q(w, \delta)$). The accuracy and confidence parameters specify the error tolerance that is acceptable in the answer. So a query with accuracy w, confidence δ and cardinality k, requires that on expectation δk of the reported values will fall within $\pm w$ of the actual values, i.e. the values that we would get if we sampled all locations. Depending on the values of w and δ, a query can range from being very loose to very strict in terms of accuracy. The smaller the window, and the higher the confidence, the stricter the query becomes, demanding more accurate results.

To answer queries in a sensornet setting using information residing in the network, we introduce *in-network summaries.* An in-network summary is a spanning tree of the network, in which every node stores a model of the data in each of the subtrees it points to. As the subtrees become smaller in the lower levels of the hierarchy, the models become finer and more precise. A query using that hierarchy can explore the structure starting at the root and going only as deep as is necessary to provide answers of good quality. We want to optimize this tree structure with the objective of minimizing the communication cost of answering queries. The guarantee we want to achieve is that for n data nodes the query will produce on expectation δn answers that are within w distance of their actual value.

Definition 7 *An* in-network summary *(or spanning summary tree) over a network graph $G(V, E)$ is a spanning tree $T(V, E')$, $E' \subseteq E$, augmented with models M_v, $\forall v \in V$. M_v is stored in node p that is the parent of v, i.e., $(p, v) \in E$.*

We are given a network graph $G(V, E)$ and a query workload $W = \{Q_i(w_i, \delta_i)\}$. We will use M_v to symbolize the model information kept for node v. Then the optimization problem of finding the optimal in-network summary can be defined as follows:

Given: Graph $G(V, E)$, query workload $W = \{Q_i(w_i, \delta_i)\}$

Find: Tree $T = G'(V, E')$ and models M_v, $\forall v \in V$, such that the average communication cost required to retrieve values to respond to $Q_i \in W$ is minimized

Our design choices for the *in-network summaries* should preserve the following requirements:

Compactness: Due to storage limitations at the sensor level summary models M_v need to be small.

Informativeness: The summaries should be as informative as possible so that queries can be answered by accessing as few of them as possible.

Accuracy: The summaries should not lead to inaccurate query answers, i.e. the confidence bounds returned by examining summaries should correctly reflect the probability of the answers falling within the accuracy window.

5.3 Model Compression

In this section we focus on one part of the in-network summary problem: the construction of models at each node. For this discussion, we assume temporarily that the structure of the tree T is given, and we want to pick the best model M_v, $\forall v \in T$. We will revisit the structure of T in Section 5.5.

Various models could be maintained in a spanning summary tree, but in keeping with prior work we assume a gaussian (normal) model. For the modeling of input *readings*, gaussian models are typically appropriate, since they successfully capture the nature of noisy measurements of physical phenomena [8]. Therefore, for all leaves in T the model M_v will be a gaussian distribution based on observations of that node's measurements. Now, when a data summary needs to represent multiple sensors, a natural extension is to use gaussian mixtures, which can be of restricted size to comply with the storage limitations of sensor nodes. A gaussian k-mixture refers to a mixture of k gaussians.

Assuming for simplicity that T has fixed fanout F, data resides at the leaves represented by the single gaussian distributions, and every internal node keeps F pairs of (childptr, gaussian k-mixture), then our "data structure" closely resembles a database index. Given a specified spanning tree T, we can imagine "bulk loading" the models M_v with a bottom-up construction, combining gaussian mixtures as we climb up the hierarchy. In this approach, mixtures high in the tree will be quite large. Since we need to keep the size restricted, we need to have a method for compressing the mixtures, otherwise called "collapsing".

In the problem of compression, our input is a mixture (set) of l gaussian distributions, and our output a k-size mixture, $k < l$. The parameter k is dictated by the amount of storage space assigned to the model on every node, and is not required to be the same on each one of them. We will first address the problem for the case of $k = 1$, assuming that the summary hierarchy keeps a single gaussian distribution as a model of all the sensors in each subtree. In Section 5.5.5 we revisit the compression model and explore generalizations to larger k.

During query execution, the only information that we have for a certain subtree is its collapsed distribution. In the case of a single-gaussian compression model ($k = 1$) the answers that this summary in the tree can provide is to report the mean μ of the summary as the answer for all nodes represented by the subtree. Intuitively, our goal is:

Given: A set of distributions $\mathcal{S} = \{N(\mu_i, \sigma_i^2)\}$

Find: Distribution $N(\mu, \sigma^2)$ that maximizes informativeness and accuracy over the original set $\mathcal{S}$.

To understand *informativeness and accuracy* in the above definition, some intuition is useful. A common approach in collapsing gaussian mixtures is to minimize some distance function (e.g. KL divergence) of the collapsed distribution from the original mixture. While intuitively this might seem to capture the quality of the compression, such an approach can actually lead to very bad decisions for our problem.

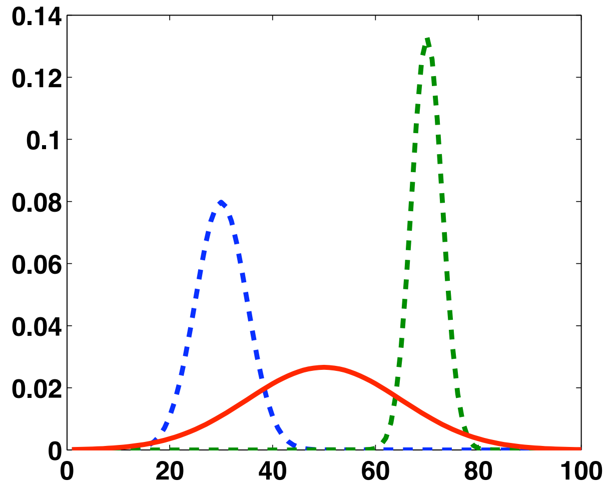


Figure 5.1. Two distributions (dashed lines) representing values of 2 sensor nodes, with no overlap. Collapsing using KL divergence produces a distribution (solid line) with significant mass in an interval that the original distributions contained almost none.

In the example of Figure 5.1 we want to collapse the 2 distributions depicted by dashed lines into one. With KL divergence as the optimization criterion, the result would be the solid line in Figure 5.1. The resulting distribution minimizes the distance from the original ones, but has the following pitfall: it falsely reports some significant mass on the interval $[45 - 60]$ where the original distributions contained almost none. In our setting this can in some cases lead to false query results for both of the original sensors involved, and thus loss of *accuracy*. A query with a large window may not suffer from this side effect, but one with a window small enough to fit in the problematic interval (e.g. width 10) could produce erroneous results.

The important observation here is that whether the answer would be wrong – and how wrong – depends on the specific query. This observation suggests that the collapsing be targeted to a specific query workload. This is the approach we follow below.

Alternatively, to avoid loss of accuracy, we could adjust the variance of the collapsed gaussian so that it will not contain any "fake" mass. But such an approach would result in an extremely wide and flat distribution. Such high uncertainty would be useless in query execution, as this summary could not produce answer estimates that would fall in a plausible query's accuracy window. In this case we have loss of *informativeness*.

5.3.1 Simple Collapsing

During compression we want to preserve as much information from the original distributions as possible. This means that the new distribution should contain as much "real mass" as possible, and this will happen if it is centered at the location that contains the most mass from the underlying distributions. This location however depends on the window in which we compute the mass. In Figure 5.1, if the window used is fairly large, then the location z that maximizes the total mass will be centered somewhere in between the two original distributions. On the other hand, if the window is small, then z would be centered at the narrowest of the original distributions. Therefore, compression of a given set of distributions will depend on the window assumed.

This also relates to query answers. In order to better understand the collapsing requirements, we need to look at how the collapsed distribution will be used to answer a given query. Assume a collapsed distribution $N(\mu, \sigma^2)$. $Q(w, \delta)$ is a query with error allowance inside a window w and confidence requirement δ. We consider that model distribution N can satisfy Q, if the mass of N in the interval $[\mu - w, \mu + w]$ is greater or equal to the confidence requirement δ, i.e. $M_{[\mu-w,\mu+w]} \geq \delta$. In that case the query reports value μ for both nodes. Note that the definition of query satisfaction is based on the belief that collapsed distribution N is an accurate representation of the underlying measurements. If N is a bad model, then any results based on it would simply be faulty. Based on our definition of query satisfaction, we need to construct N in such a way so that $M_{[\mu-w,\mu+w]} \geq \delta$ is true not only for N, but also for the uncompressed mixture.

As it is obvious from Figure 5.1, if we choose N as depicted by the solid line, then for small values of w the query may apparently pass the mass test, but the response will be wrong, as neither of the original nodes has value $\mu \pm w$ with δ confidence. However, if the window w is large enough, the distribution N may be sufficient to answer Q without problems. In order to make sure that that answer will be correct based on the query requirements we need to make sure that the mass of the collapsed distribution in the interval $[\mu - w, \mu + w]$ is the same as the total mass of the original distributions in the same interval. This observation leads directly to a simple approach for collapsing.

Our two requirements for collapsing are that it should not introduce "fake mass" (high accuracy), and it should retain as much "real mass" as possible (high informativeness). This makes sense for a specific choice of window, and it guarantees that queries of the same window will get accurate response. Formally, the problem we want to solve is as follows:

Given: One dimensional function f representing a probability distribution (in our case a gaussian mixture)

Find: z s.t. the mass of f in $[z - w, z + w]$ is maximized.

$$\max_z \int_{z-w}^{z+w} f(x)dx \tag{5.1}$$

Location Of Maximum Mass

The solution to the maximization given in equation (5.1) cannot always be determined analytically. One approach is to numerically evaluate it through the use of sliding windows: given window w, "slide" the interval $[z_i - w, z_i + w]$ across the x-axis calculating the mass of the distribution for every location z_i and pick the one with the maximum value. Obviously the quality of the result will depend on the fineness of the discretization $D = \{z_i\}$ used for the sliding.

Another approach is to use gradient ascent starting at the means of the original distributions. This is because for continuous functions, the optimal location of the window is required to contain at least one local maximum.[1] However this approach is not guaranteed to find the optimal solution either, and one can construct adversarial examples where that happens. In Figure 5.2 we empirically compare the two approaches of sliding window and gradient ascent, which are shown to perform equivalently.

For this experiment we used real data from the Intel Berkeley Lab deployment [28] to compute gaussian models for 54 nodes, and used them to compare the two algorithms for different window sizes. The discretization used for the sliding window algorithm was

[1]It is easy to show that if it does not contain a maximum, sliding the window to one direction would increase the contained mass.

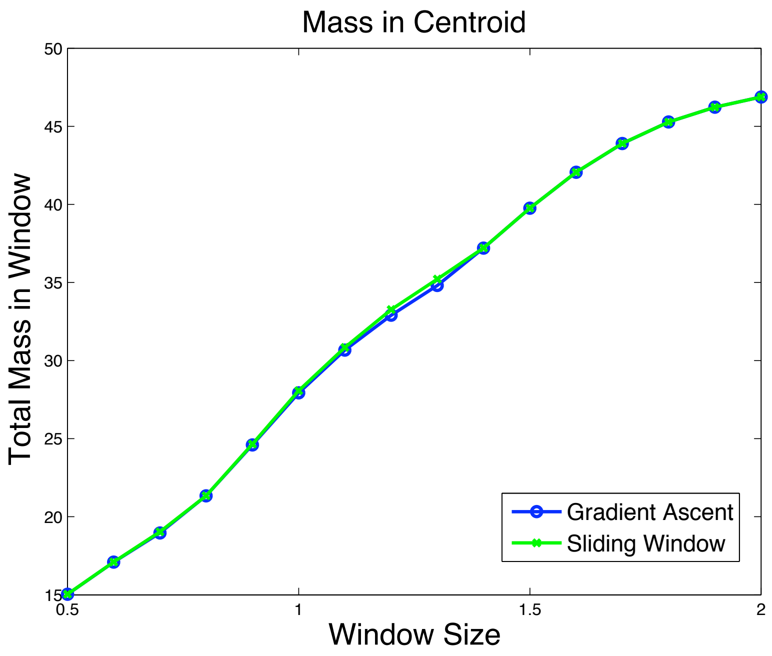


Figure 5.2. Comparison of the sliding window and gradient ascent algorithms

a $D = \{z_i\}$ where $z_{i+1} - z_i = 0.001$ In conclusion, the sliding window technique with a modest discretization produces equivalent results as gradient ascent, when at a much lower computational cost.

5.3.2 Tail-aware Collapsing

Note that the compression algorithm described in Section 5.3.1 suffers from accuracy problems in the case of recursive collapsing: by recursively applying simple collapsing to compress distributions with one another, we run the risk of introducing “fake mass” and thus reducing accuracy. Even though simple collapsing does guarantee that the mass inside window w of its mean is accurate (i.e. corresponds to real mass from the original distributions), there is no guarantee for parts of the interval $[\mu - w, \mu + w]$. What that means is that even though it is guaranteed that the mass of the result distribution in $[\mu - w, \mu + w]$ is going to be exactly the same as for the original distributions, the guarantee does not hold for any interval $I \subset [\mu - w, \mu + w]$.

The same is true for the tails of the distribution: the intervals $[-\infty, \mu - w] \cup [\mu + w, \infty]$.

Recursively collapsing distributions can therefore cause errors due to the introduced fake mass.

Even though the mass inconsistencies inside subintervals of $[\mu - w, \mu + w]$ are not simple to deal with, the possibly problematic tails are easy to fix with *tail-aware collapsing*. Tail-aware collapsing performs compression exactly the same way as simple collapsing, but simply disregards any mass that exists in the tails of the distributions. Note however, that this approach can err in the opposite direction: it avoids the introduction of fake mass through the tails, but may disregard real mass that may have actually existed in the tails. Therefore, compared to simple collapsing, tail-aware is more conservative.

5.4 Query Traversal

In the previous section we discussed optimal compression of a set of input distributions based on an expected query workload. Sensor nodes organized in a tree structure can create an in-network summary by recursively performing compression bottom up, from leaves to root. In this section, we focus on the problem of routing queries using an in-network summary, making routing decisions at each node based on the local model M_v. First we discuss what the optimal traversal would be for query $Q(w, \delta)$ on tree $T = G(V, E)$. A traversal is a connected component of G that contains the basestation. Since G is a tree, every connected component in it is also a tree. The optimal traversal is that of minimum total cost that *satisfies* Q on expectation (Def. 8).

Definition 8 (Query Satisfaction) *In a network of n nodes, a response $R = \{r_1 \dots r_n\}$ to a query $Q(w, \delta)$ is said to satisfy Q if on expectation the actual values of at least δn nodes fall in their respective interval $[r_i - w, r_i + w]$.*

Due to linearity of expectations, assuming that the underlying distributions are correct, then the query will be satisfied if $\sum_i \int_{r_i - w}^{r_i + w} f_i(x)dx \geq \delta n$. So the optimal traversal problem can be defined as follows:

Given: tree $T = G(V, E)$ and node models M_v, $\forall v \in V$

Find: $G'(V', E')$, $E' \subseteq E$, such that $\sum_e Mass(M_u, w) \geq \delta n$, where $e = (u, v)$ with $u \in V'$ and $v \in V \backslash V'$

In the above problem statement, $Mass(M_u, w)$ is the maximum mass that can be contained inside window w from model M_u. If the mixture model is compressed to a single gaussian, $Mass(M_u, w) = \int_{\mu-w}^{\mu+w} f(x)dx$, where f is the normal probability density function.

5.4.1 DP Traversal

We will solve the optimal traversal problem using a dynamic programming algorithm. Every node will keep a DP table (in our case just a vector) which will hold information on forwarding decisions based on assigned budget. For example, if v is the root of subtree T_v and C_{T_v} is the cost of traversing the entire subtree T_v, then v will keep a vector of length C_{T_v} where every entry will be the maximum mass that can be collected by assigning the corresponding budget to that subtree. For a tree of fanout F, children $u_1, \ldots, u_F$ of node v and a budget assignment $B = \{b_1, \ldots, b_F\}$ among the F children, the DP function for computing the entry $J_v(c)$ will be:

$$J_v(c) = \max_B \sum_{i=1, b_i>0}^{F} J_{u_i}(b_i) + \sum_{i=1, b_i=0}^{F} Mass(M_v, w)|T_{u_i}| \tag{5.2}$$

where $|T_{u_i}|$ corresponds to the number of nodes represented by the subtree of node u_i. The second summation in (5.2) gives a mass estimate for the unvisited children ($b_i = 0$) based on the model of node v. Also, $\sum_i b_i = c - \sum_{i \ s.t. \ b_i>0} w_{(v,u_i)}$, where $w_{(v,u_i)}$ the weight of the edge (v, u_i), and the sum subtracted in this formula adjusts the available budget by the cost of reaching those children of v with non-zero budget assignments. The weight of each (v, u_i) edge can be simply defined as the number of hops needed to reach that child. Along with the DP vector, the choice of best budget assignment for each cell should also be kept. At the leaves of the tree, the DP vector is a single element, $J(0) = Mass(M_v, w)$.

Algorithm 6 outlines this dynamic program for the case of a binary tree ($F = 1$), and unit edge weights. The code can be easily extended to the more general case of fanout F or even unrestricted fanout.

With the DP in place, routing decisions can easily be made depending on the δ parameter of the query. Given δ we can compute the desirable mass as δn, where n is the total number of nodes in the network. Using the DP table at the root, we can find the smallest budget that achieves that mass, say b_i, and the values of $leftBudget(b_i)$ and $rightBudget(b_i)$ will give the budget assignments for the left and right child respectively. The traversal descends until the budget is exhausted. The DP vector for each node is equal to the number of nodes

Algorithm 6 DPConstruct(v,w,B)

```
 1: if $B < 0$ then
 2:     return 0
 3: end if
 4: if $B == 0$ then
 5:     $J(B) = Mass(M_v, w)$
 6: else
 7:     for $k = 0 \dots B$ do
 8:         lMass(k) = DPConstruct($u_1, w, k - 1$)
 9:         rMass(k) = DPConstruct($u_1, w, B - k - 1$)
10:     end for
11:     $J(B) = \max_k(lMass(k) + rMass(k))$
12:     $leftBudget(B)$ =argmax$_k(lMass(k) + rMass(k)) - 1$
13:     $rightBudget(B) = B - leftBudget(B) - 1$
14: end if
```

in that node's subtree, so if n are all the nodes in the tree, the space needed would be $O(n)$. Constructing the DP vector for each node has complexity $O(n^2)$, so the whole algorithm has complexity $O(n^3)$.

The described DP approach can compute the optimal traversal solution for a query of window w, on a tree model compressed using the same window. One problem is that the DP tables can become large at nodes high in the hierarchy, which could violate our limited storage principle.

As an alternative, we proceed to propose a simple greedy traversal algorithm that makes decisions locally at every node, without the requirement of keeping extra information, like the DP tables.

5.4.2 Greedy Algorithm

Our greedy descent algorithm is quite straightforward. The query is initiated at the root of the tree, and every node decides whether to descend or not based on the satisfiability of the query by the local model:

$$\int_{\mu-w}^{\mu+w} f(x)dx \geq \delta \tag{5.3}$$

If the model at the current node satisfies (5.3), then no descent is necessary. Otherwise the query is forwarded to the node's children. In this simple version of the algorithm, if a decision is made to forward, then all of the children will receive the query. More elaborate schemes can give different priorities to children and perform selective forwarding. This however would require extra communication, as decisions to forward might depend on traversal results on other subtrees, and cannot be made only locally.

Algorithm 7 GreedyDescend(node$_i$,w,δ)

```
1: Compute $I = \int_{\mu-w}^{\mu+w} f(x)dx$
2: **if** $I \geq \delta$ **then**
3:     return $\mu_i$
4: **else**
5:     GreedyDescend(children$_i$,w,$\delta$)
6: **end if**
```

Algorithm 7 is more conservative than the DP approach, as it applies the satisfiability definition on every subtree and not just the global tree. The algorithm will terminate the recursion at a set of nodes each of which satisfy the query according to Definition 8 in their local subtree. If k_i is the number of nodes represented by every subtree i where the recursion terminated, then on expectation $\delta \sum_i k_i = \delta n$ have accurate within w values, which in turn means that globally the query is satisfied. So every solution of the greedy algorithm is guaranteed to satisfy query Q, but the satisfiability definition only requires it to hold globally and not for every subtree, making Algorithm 7 conservative.

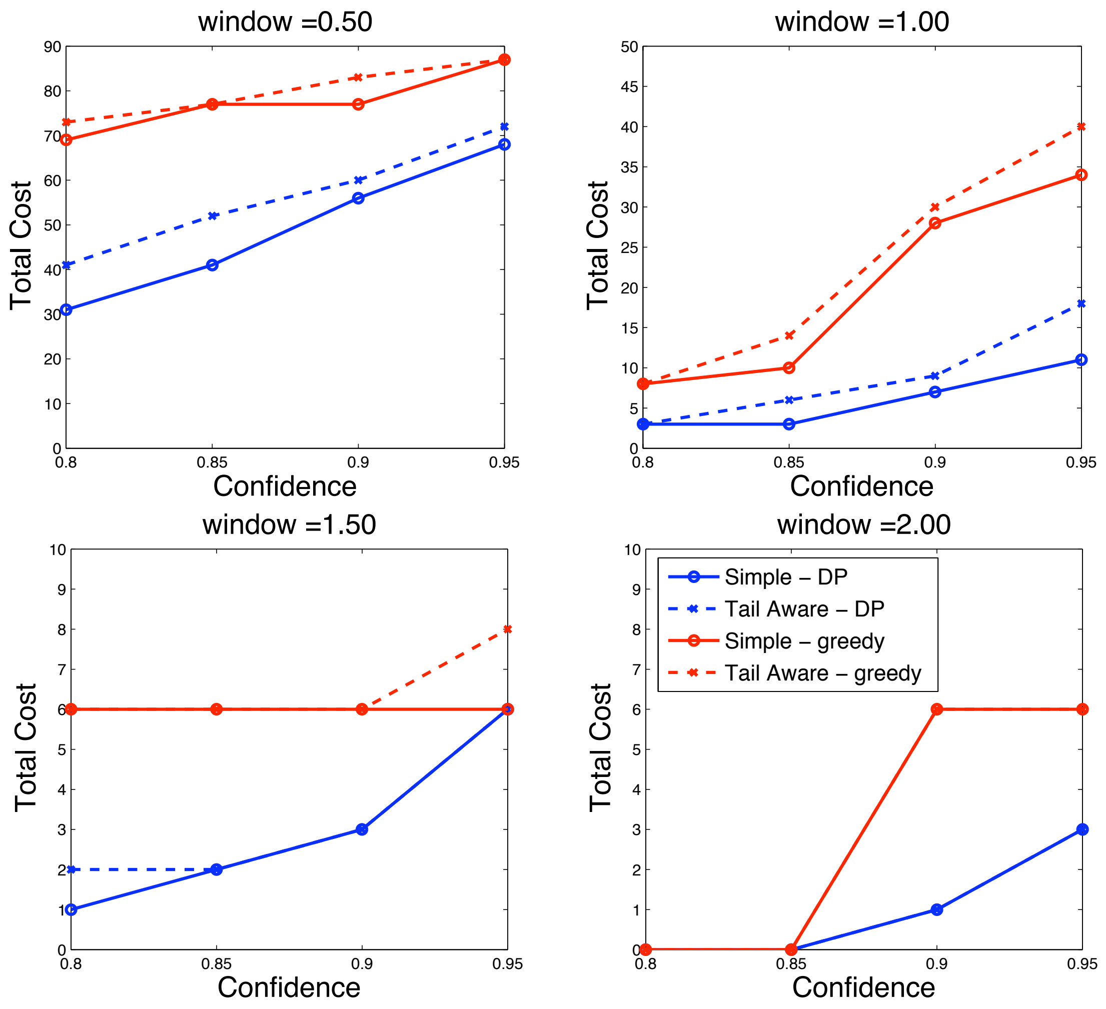


Figure 5.3. Evaluation of cost of Greedy against optimal cost found by the DP algorithm. The "Simple" and "Tail-aware" schemes refer to the type of compression deployed (Section 5.3)

We evaluate the performance of the greedy algorithm, comparing it with the DP solution for different sizes of the window w, and against different values of the confidence parameter δ depicted on the x-axis. Both algorithms in this case are executed on the same binary tree with data gathered from the Intel Berkeley Lab deployment [2]. To construct the initial leaf distributions, data from one hour is analyzed, and nodes are grouped together in the tree based on spatial proximity. The results are given in Figures 5.3 and 5.4. The greedy algorithm demonstrates a more conservative behavior, resulting in a smaller number of reported errors and higher cost. It still remains a competitive alternative, without requiring maintaining state in the nodes as the DP approach does.

In this experiment, the choice of range of window sizes for the query workload ([0.5,2]) relates to the variance of the underlying data. Queries with windows smaller than 0.5 would always sample all, or most, locations, as the queries are now too strict even for the leaf level models. Also, experiments with very big windows provide no useful information, as the queries become so loose relative to the data that even the rough root level model is enough to satisfy them.

In Figure 5.5 we demonstrate the benefits of our compression scheme, based on mass maximization, with KL divergence based compression. Depicted are the results for a specific query window size, but the rest behave similarly. The comparison is done using both the DP and greedy traversal algorithms, and in both cases our compression results in plans of lower cost. An important note is that the temperature data used for our experiments actually gives KL divergence some advantage, and yet it still loses. With data points whose underlying distributions differ a lot, the problems of the KL based compression would become more exaggerated.

5.5 Tree Construction

In the previous sections we discussed methods for performing compression and query traversal. However, the algorithms were applied over a predefined tree, and even though compression is optimally done given a specific design workload (i.e. a set window), the way the nodes are grouped affects the quality of compression. Grouping of dissimilar nodes into the same subtree will inevitably lead to bad compression, and therefore bad performance. In this section we will discuss the problem of finding the optimal tree for a specific query workload.

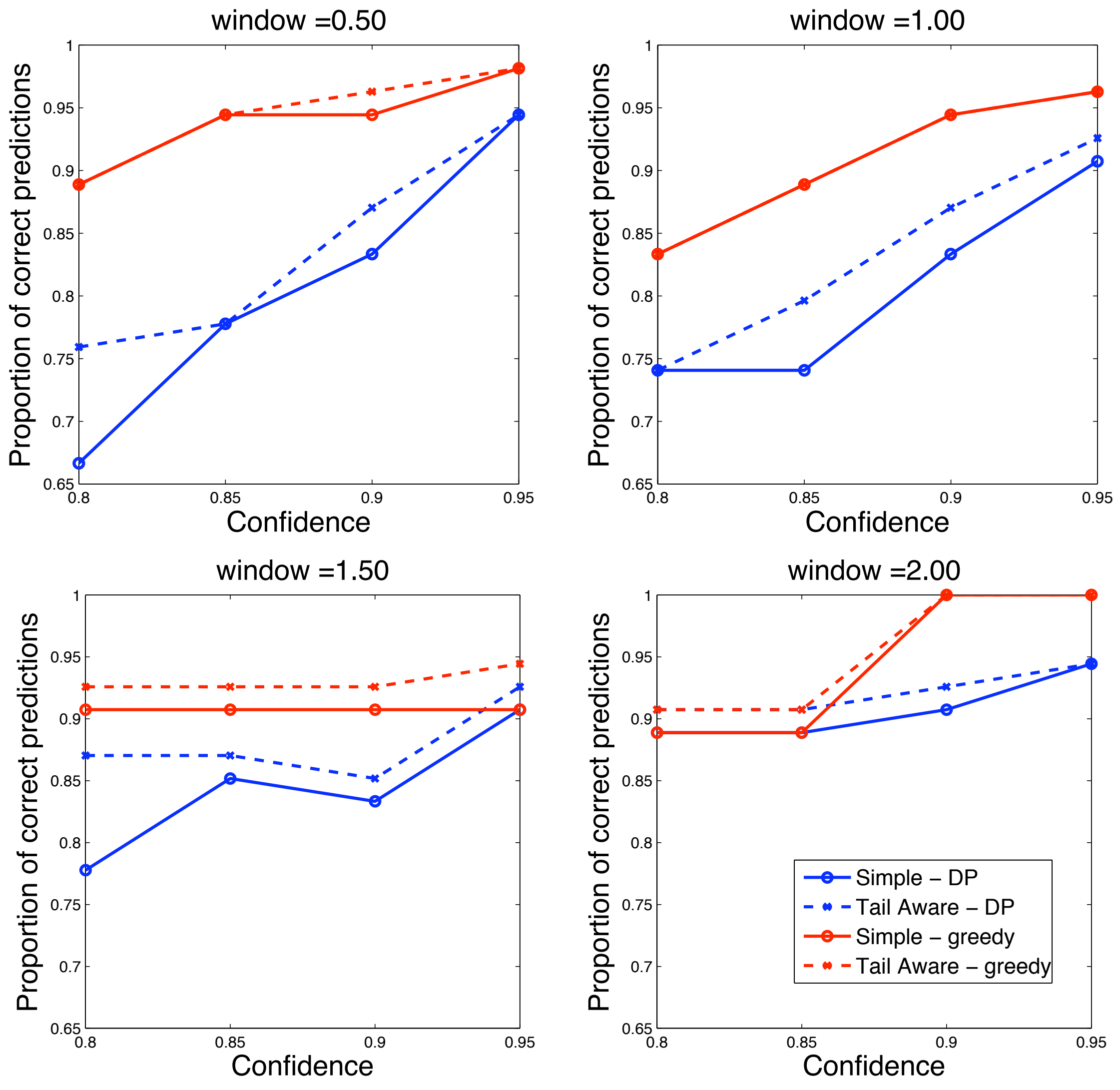


Figure 5.4. Comparison of the proportion of correct responses for the greedy and the optimal cost traversal chosen by the DP algorithm. The "Simple" and "Tail-aware" schemes refer to the type of compression deployed (Section 5.3)

We focus on the case of graphs with unit edge weights. From a practical perspective, a communication link between two nodes is considered to exist if the nodes have a packet loss rate bounded by a threshold.

The metric to minimize is communication cost during query execution. Assume T_{OPT} is an optimal tree that produces the minimum possible cost when traversed by query $Q(w, \delta)$. A traversal of T_{OPT} using the greedy algorithm (Alg. 7) will stop at nodes on various levels

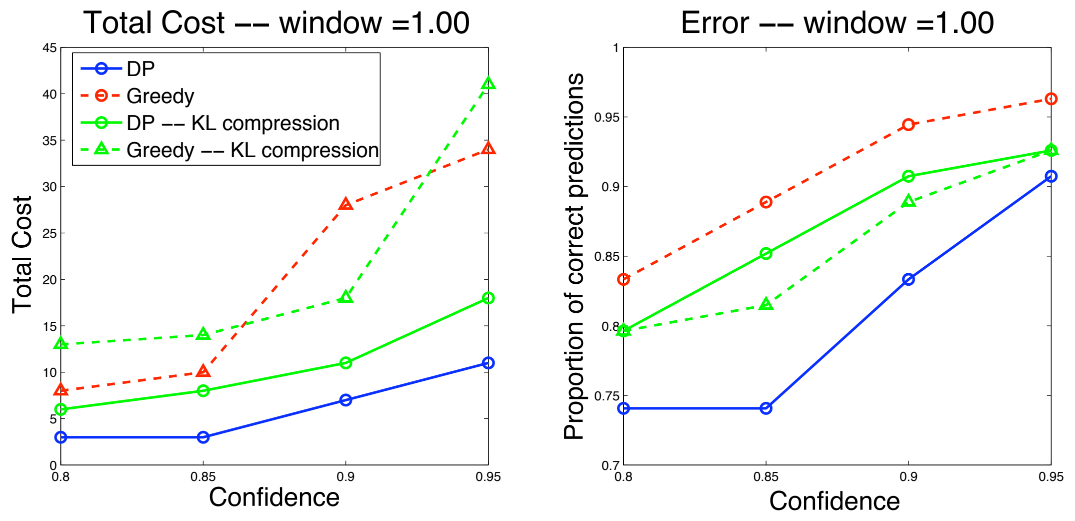


Figure 5.5. Comparison of our compression method with KL divergence based compression, using DP and greedy traversal

that all satisfy inequality (5.3). Define the cut C as the set of nodes at which Alg. 7 stops. Note that there is no path from root to leaf in tree T_{OPT} that does not "hit" the cut C. The structure of the tree below the cut, i.e. all nodes that have an ancestor in C, is irrelevant to the cost of query Q, as the query traversal will never descend that far.

Also, in the case of trees with constant fanout, the structure above the cut is irrelevant as well as shown in Theorem 8:

Theorem 8 *In a tree with fixed fanout F, the cost to traverse from the root to cut, i.e. the cost to reach all nodes in the cut from the root, is $\frac{F}{F-1}(|C|-1)$, where $|C|$ is the size of the cut.*

Proof: First we will prove that the size of a cut C_k is $|C_k| = (k-1)F - (k-2)$, where k integer. What this implies is that depending on the fanout, cuts cannot be of any size. For example a tree of fanout 3 cannot have a cut of size 2.[2] In this discussion C_k will be a cut of size order k. This doesn't mean that C_k is of size k, but rather that it is one order bigger than a cut of order $k-1$. The exact size as we will shortly prove is given by

$$|C_k| = (k-1)F - (k-2) \tag{5.4}$$

The proof will be done by induction.

[2]Remember that with the greedy algorithm, two nodes that are in the same cut cannot be in an ancestor-descendant relationship, as the greedy algorithm forwards the query to either all children or none

- For k=1 the cut only contains the root and it has size $|C_1| = (1-1)F - (1-2) = 1$.
- Assume that for order k a cut C_k has $(k-1)F - (k-2)$ nodes.
- We will now show that a cut C_{k+1} has $kF - (k-1)$ nodes:
  Every cut of order k can be transformed into a cut of order $k+1$ by replacing a node in the cut by its F children. That would increase the cardinality of the cut by F. Since all cuts of order k have the same number of nodes, then all cuts of size $k+1$ that are constructed this way will also have the same number of nodes. The only way that a cut C_{k+1} may not have the same cardinality would be if there is no cut of order k that can produce C_{k+1} with the above method. That means that C_{k+1} does not contain any F nodes that are siblings, otherwise these could be replaced by their parent to form a cut of order k. Now let us see if that is possible. Assume that node $v \in C_{k+1}$ is the deepest node in the cut, i.e. the path from root to v is the longest path from root to the cut. v's siblings is nodes $u_1 \dots u_{F-1}$. If $\exists u_i \notin C_{k+1}$ then this means that there is a node in C_{k+1} that is deeper than node v because there is no path from root to leaf that does not cross C_{k+1}. This is impossible because we assumed that v is the deepest node in the cut. Therefore $u_i \in C_{k+1}$ and thus there is a cut of order k that can produce C_{k+1}. That means that C_{k+1} has the same cost as any other cut of order $k+1$. Now, since $|C_{k+1}| = |C_k| + (F-1)$ we can derive that $|C_{k+1}| = kF - (k-1)$. QED

Using the exact same type of induction we can show that the cost to traverse to a cut of order k is $(k-1)F$. Solving equation (5.4) in terms of k and replacing to the cost formula, we get that the cost is equal to $F\frac{(|C|-1)}{F-1}$. ■

5.5.1 Optimal Tree Problem

Theorem 8 shows that if C is the cut that Alg. 7 picks for query $Q(w, \delta)$, then the structure of the hierarchy above and below the cut are irrelevant to the cost of answering the query, and therefore only the size of the cut determines query cost. A node in the cut acts as a representative for the whole subtree, so the cut is a set of connected components each of which contains a representative node with model M_v that satisfies Q.

The Optimal Tree Problem with respect to query $Q(w, \delta)$, is the problem of finding a tree that satisfies Q using Alg. 7 with minimum communication cost, and is formally defined as follows:

Definition 9 (Optimal Tree Problem)

Given: *Graph $G(V, E)$ with the cost function $c : E \rightarrow \mathcal{R}_+$ and query $Q(w, \delta)$*

Find: *Set of components $S = \{C_1 \dots C_k\}$ such that*

- *$\forall i$ C_i is connected in G.*
- *$\forall i$ $\frac{1}{|C_i|} \sum_{j \in C_i} Mass_j(C_i) \geq \delta$.*
- *$\cup C_i = V$ and $C_i \cap C_j = \emptyset$ for $i \neq j$*
- *With the objective to minimize the cost of the minimum cost subtree T of G that contains at least one vertex from each component C_i.*

In the above definition $Mass_j(C_i)$ represents the total mass that node j contributes to component C_i.

The objective of the Optimal Tree Problem (4th bullet) is the Group Steiner Tree Problem: given a graph $G(V, E)$, cost function $c : E \rightarrow \mathcal{R}_+$ and sets of vertices $g_1, g_2, \dots, g_k \subset V$, find the minimum cost subtree T of G, that contains at least one vertex from each set g_i.

Since the Group Steiner Tree Problem is NP-hard, the Optimal Tree Problem is also hard. The GST problem has a polylogarithmic approximation [35], so we will attempt to address the selection of components C_i as a separate problem.

5.5.2 Optimal Clustering

From Definition 9, the division of nodes into the components C_i can be viewed as a clustering problem. Each cluster corresponds to a subtree rooted at a node chosen by the query cut. Each cluster is of limited diameter (Def. 9, 2nd bullet) and, according to Theorem 8, we want to find the minimum size cut for the specific query, so equivalently the optimal clustering should minimize the number of clusters.

This subproblem is also hard, but we provide a greedy algorithm that approximates the optimal solution by a logarithmic factor. The pseudocode is given in Alg. 8.

Algorithm 8 GreedyClustering(V, w, δ)

```
1: $Clusters = \emptyset$
2: Pick discretization $D = \{z_1, \ldots, z_k\}$
3: **repeat**
4:   **for** $z \in D$ **do**
5:     $S_z = \emptyset$
6:     **for** $v_i \in V$ **do**
7:       $Mass_z(v_i) = \int_{z-w}^{z+w} f_i(x)dx$ //$f_i$ the model of node $v_i$
8:     **end for**
9:     **while** $\sum_{v_i \in S_z} Mass_z(v_i) \geq \delta|S_z|$ **do**
10:      $v^*$ =argmax$_i Mass_z(v_i)$
11:      remove $v^*$ from $Mass_z$
12:      **if** $\sum_{v_i \in S_z \cup \{v^*\}} Mass_z(v_i) \geq \delta|S_z|$ **then**
13:        $S_z = S_z \cup \{v^*\}$
14:      **else**
15:        break;
16:      **end if**
17:    **end while**
18:  **end for**
19:  $S_z^*$ =argmax$_i|S_{z_i}|$
20:  $Clusters = Clusters \cup S_z^*$
21:  V = V $\setminus S_z^*$
22: **until** $V = \emptyset$
```

Proposition 1 *Algorithm 8 provides a factor* $\log(n)$ *approximation to the optimal clustering, which minimizes the number of clusters.*

Proposition 1 is easily derived when one notices that Algorithm 8 is equivalent to greedy Set Cover. In practice the algorithm behaves a lot better than this guarantee, with results always close to the best solution. The graphs from those experiments were omitted due to space constraints.

It is important to note that Algorithm 8 may violate one of the conditions of Definition 9, that each cluster has to be connected. Greedy Clustering greedily adds nodes to each fixed cluster center from the discretization. For a fixed center, the weight of a vertex is constant and independent of which other vertices join the same cluster. Without the connectivity requirement, the greedy addition of vertices in decreasing weight will indeed create the cluster with the largest cardinality for that interval. However, if connectivity is enforced, the problem becomes NP-hard as we proceed to show.

Maximum Connected Subgraph of Limited Diameter:

Given: Graph $G(V, E)$, vertex weight W_u for vertex u, maximum diameter D. [3]

Find: $G'(V', E')$ s.t: G' is connected, $\frac{1}{|V'|}\sum_{u \in V'} W_u \leq D$, with the objective to maximize $|V'|$

Theorem 9 *The Maximum Connected Subgraph of Limited Diameter Problem (MCSLD) is NP-hard.*

Proof: Assume that there exists a polynomial algorithm that solves MCSLD. We will show that we would then be able to solve any instance of Set Cover in polynomial time.

Assume an instance of Set Cover, with a set of sets $\mathcal{C} = \{C_1, \ldots, C_k\}$. Each C_i is a set containing some elements $\{s_j\}$. We will transform it into an instance of MCSLD as follows: Construct a graph G by creating a node for each C_i, a node for each element s_i and one extra node, let's call it H. Connect H with all the C_is. Connect each C_i with all the elements s_i that are contained in C_i. Give all the s_i and vertex H a vertex weight of $\frac{1}{n+1}$, where n the total number of elements s_i. Assign to each C_i a vertex weight of $A >> 1$.

[3]With vertex weight W_u being the mass of u's distribution outside the window interval, then D would be $1 - \delta$

We will call the MCSLD algorithm on graph G with $D = 1 + A$. If the algorithm returns a solution, then this is equivalent to a set cover of size 1, as the only way to get a connected component with total cost at most $1 + A$ is to include exactly one of the C_i. It is equally easy to see that when MCSLD is called with $D = 1 + rA$, its solution is equivalent to a set cover of size r. Therefore calling MCSLD at most k times, which is polynomial in terms of the input of Set Cover, would give us the optimal solution to Set Cover. But since Set Cover is NP-hard, then MCSLD has to be hard as well. ■

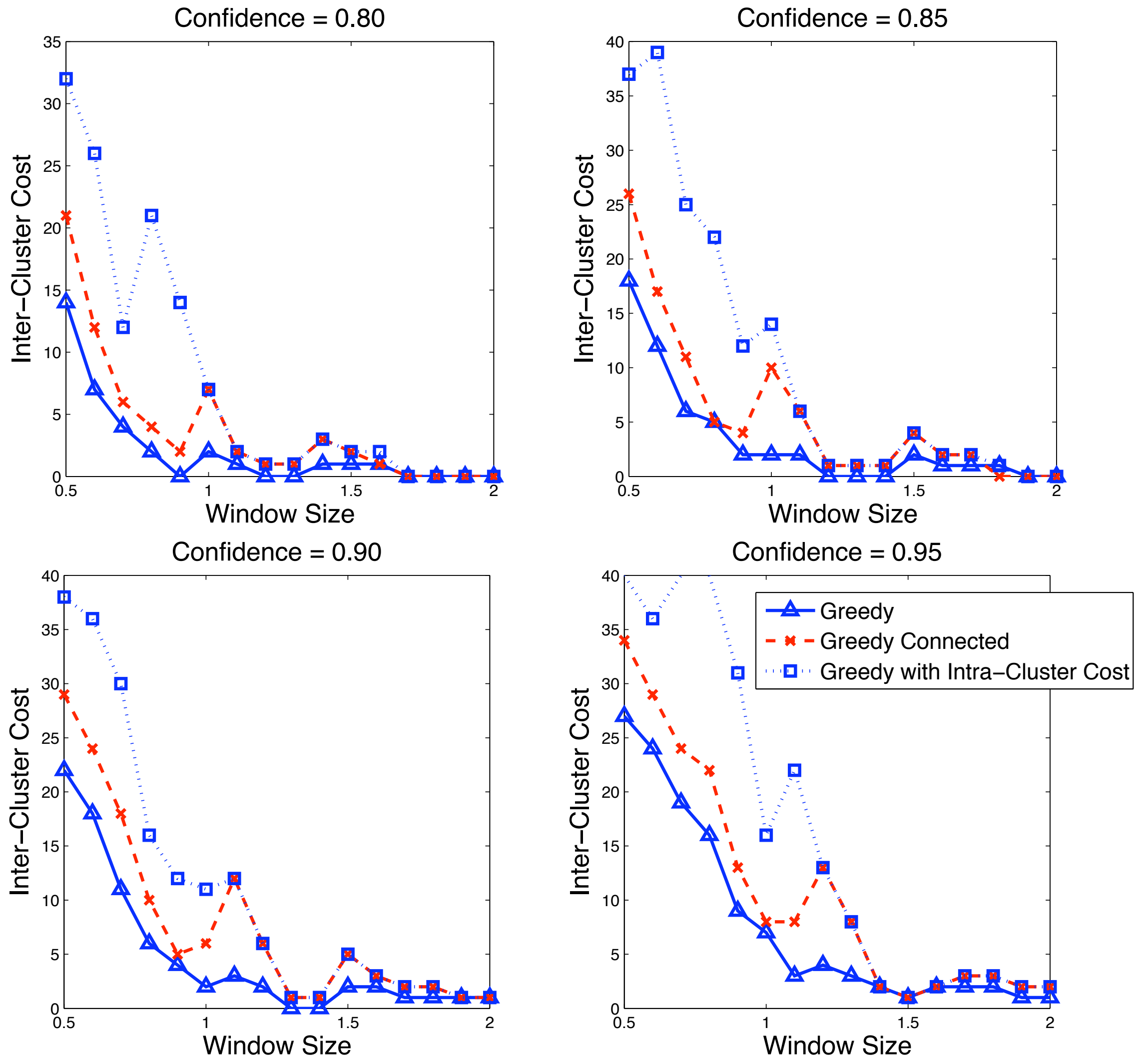


Figure 5.6. Comparing the different clustering approaches, based on the communication cost for varied parameters of window size and confidence for the query workload.

The connectivity requirement ensures that the participating nodes can form a tree. To enforce connectivity we can either: (a) augment the clusters chosen by Algorithm 8 with extra communication nodes to force connectedness, or (b) change the greedy algorithm so it only augments the clusters with nodes accessible by those already in the cluster. With option (b) the algorithm will no longer have the logarithmic guarantee. Option (a) may be more appealing in most cases, as the extra cost for intra-cluster communication is only setup cost for the cluster formation and determination of the common model, and does not inflict extra cost during query time. Figure 5.6 presents a comparison of communication cost between clusters constructed by the different approaches. The numbers for Greedy with Intra-Cluster cost encode the model setup cost; during query execution communication cost is equivalent to Greedy and therefore beats the Connected Greedy approach.

5.5.3 Distributed Clustering

Algorithm 8 provides a centralized approach in approximating the minimum number of clusters. In this section we will give a distributed clustering algorithm, and experimentally compare it with the centralized one.

Algorithm 9 Distributed Expanded Neighborhood

```
1: Basestation broadcasts clustering msg
2: Each node picks a random wait time.
3: if wait time passes without cluster requests then
4:    node initiates clustering
5: end if
6: repeat
7:    node randomly selects v from neighbor table
8:    if $d \leq D_{max}$ then
9:       node sends cluster request to v
10:      Augment neighbors table with neighbors of v
11:   else
12:      Remove v from neighbors
13:   end if
14: until no more neighbors
```

A distributed approach avoids the overhead of communicating all the data to a centralized location, and is more suitable for performing selective reclustering in case some part of the hierarchy needs to be updated. Algorithm 9 basically initiates at a single node level where a cluster of size 1 is created. The node looks up neighbors on its neighborhood table and attempts to augment its cluster size by inviting them to join the cluster. Once a new neighbor joins, the current neighborhood is augmented by the newcomer's neighbors. The cluster stops growing when no more neighbors can be added without exceeding the maximum allowed cluster diameter D_{max}. Note that the algorithm requires some bookkeeping and messages to communicate neighborhood tables. A simpler version of distributed greedy clustering uses a random walk to augment the cluster, with a node making a pick only from its own neighbors and not the complete neighborhood defined by the cluster. The cluster stops growing when the random walk cannot proceed without violating the cluster diameter.

In Figure 5.7 we compare the 2 centralized approaches, greedy and connected greedy, and the 2 distributed approaches, expanded neighborhood and random walk. The simulation experiments were performed in Matlab, with real data from the Intel Berkeley Lab deployment. The algorithms are compared on the cardinality of the clusterings that they create. Out of all the approaches, greedy is the only one that does not enforce the connectivity requirement, which does not ensure that it will give the best result as it is not optimal, but has a logarithmic approximation guarantee.

From these experiments we observe that the results of distributed clustering are comparable to the centralized ones, especially to connected greedy. On average the distributed expanded neighborhood and random walk heuristics performed within a factor of 1.4 and 2.8 respectively of the centralized greedy algorithm which has a guaranteed logarithmic factor approximation of the optimal solution.

5.5.4 Building Trees for Varied Workload

In Section 5.5 we discussed the problem of constructing the optimal tree for a specific query. We connected it with a clustering problem, we showed hardness results and gave approximation algorithms and heuristics. In this section we will extend our approach to the setting of a more varied query workload. We will assume a baseline confidence δ, and query workload that includes various different windows $\{w_1, \ldots, w_k\}$.

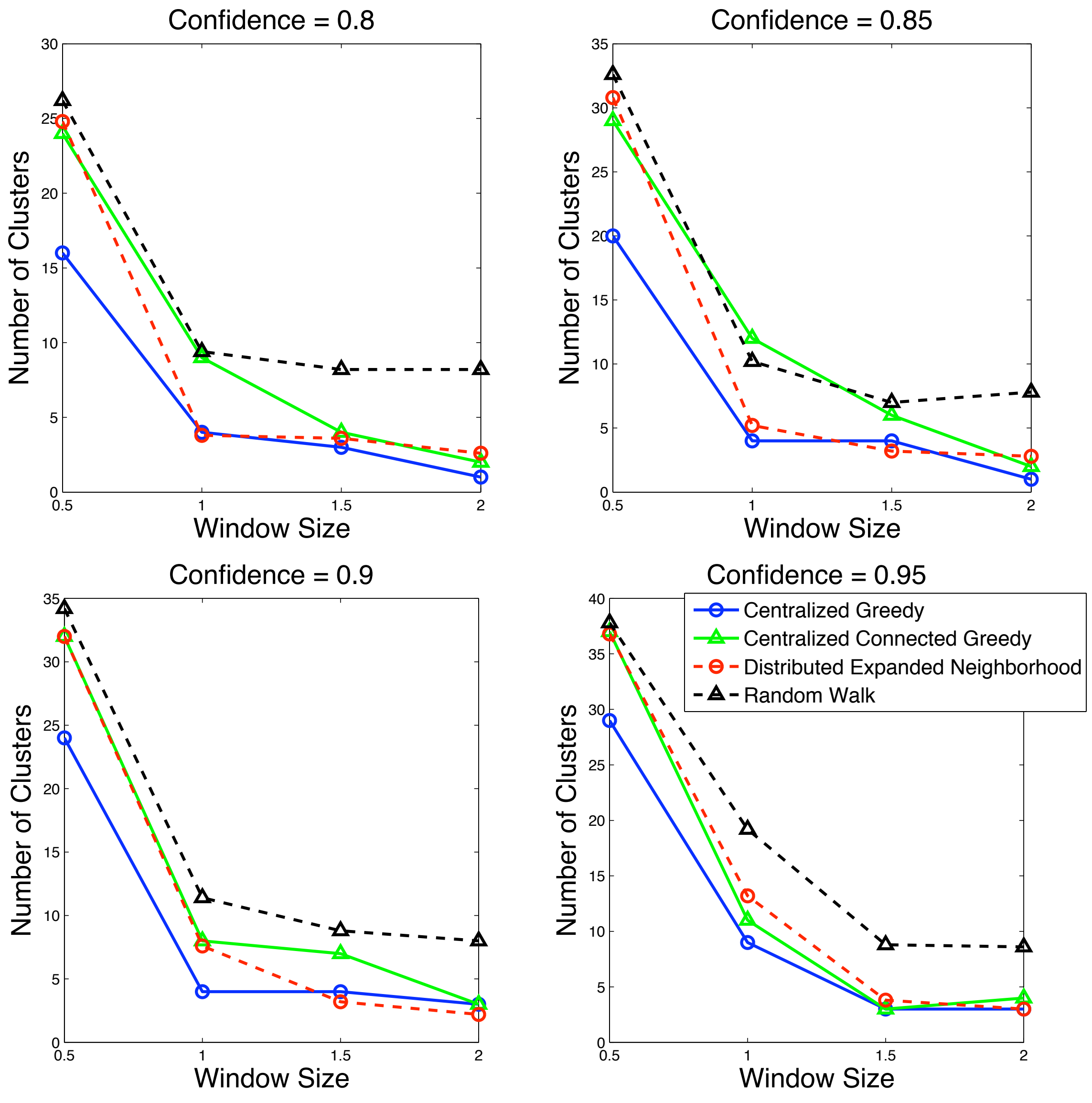


Figure 5.7. Comparison of the distributed and centralized clustering algorithms.

Following the intuition that models high in the hierarchy present a coarse view of the data, while moving into deeper levels provides more detail, it seems natural to address varied workload by recursively clustering in decreasing order of window size. Clustering will start with the largest window size which represents the less strict query in the workload. The clusters produced are further divided into smaller clusters using the next largest window in the query workload, and the process continues in that fashion until the smallest window size.

This heuristic, sketched in Algorithm 10 produces a tree in which every level corresponds to a different window size, from larger in the higher parts of the hierarchy, to smaller in the deeper levels.

Algorithm 10 TreeConstruction(G,wRange)

```
1: sort(wRange)
2: G(k+1) = G;
3: for i = k downto 1 do
4:    w = wRange(i)
5:    G(i) = cluster(G(i+1),w)
6:    Connect $G(i)$ with $G(i+1)$
7: end for
```

We experimentally evaluate the performance of our heuristic by comparing its communication cost for queries of the different window sizes, versus a tree that was geared only towards the specific window at hand. We design a single tree T for window sizes 0.5, 1, 1.5 and 2, and confidence 0.9, as well as 4 other trees each one geared towards a single one of the previous window values. The choice of window sizes is again dictated by the variance of the underlying data which we are modeling, as explained in Section 5.4.2. Compression with window sizes that are too small or too big, relative to the underlying data variance, would produce models of too high variance that would not be useful in query answering. The communication cost of the query workload over T is evaluated against the corresponding cost of the optimal tree for each window size.

Figure 5.8 shows that Algorithm 10 approximates well the best single clustering solution. Note that the tree construction is not specific to the type of clustering used, and any of the clustering algorithms that we proposed –distributed or centralized– can be used for that step.

Figure 5.9 evaluates a tree designed for a workload of window sizes 0.5, 1, 1.5 and 2, and confidence of 0.9 using Algorithm 8 over a query workload of a finer range of window sizes and varied confidence. The communication cost behavior resembles a step function, because depending on the strictness of the query (confidence and accuracy window) the traversal usually picks for the most part one level of the hierarchy. The results demonstrate

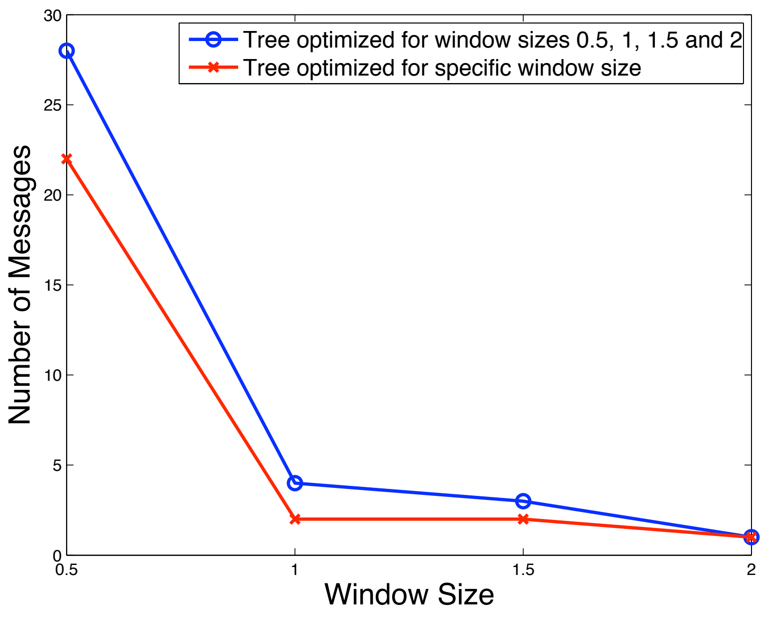


Figure 5.8. Comparing the performance of a tree designed over workload W vs a tree clustered over a single window

that the confidence parameter of the query workload does not have a considerable impact on communication cost.

Figure 5.10 compares the performance of hierarchies constructed using the tree construction heuristic (Alg. 10) for a design workload of window sizes 0.5, 1, 1.5 and 2, and confidence of 0.9, with different methods for the clustering step: centralized connected greedy (Alg. 8), distributed expanded neighborhood (Alg. 9) and random walk. The hierarchies are evaluated on the communication cost of a query workload of a fine range of window sizes. The distributed algorithms are somewhat outperformed by the centralized approach, but they still beat traditional data gathering approaches like TinyDB ([59]) which gathers data along a spanning tree. The TinyDB cost is constant and independent of the query parameters.

5.5.5 Enriched Models

Our analysis up to this point focused on Single Gaussian Model (SGM) compression schemes, which have minimal requirements of storage space from the nodes. In this section we examine more complex models and evaluate their performance against SGMs. Note that the cost involved is not only storage space, but also communication, as larger models

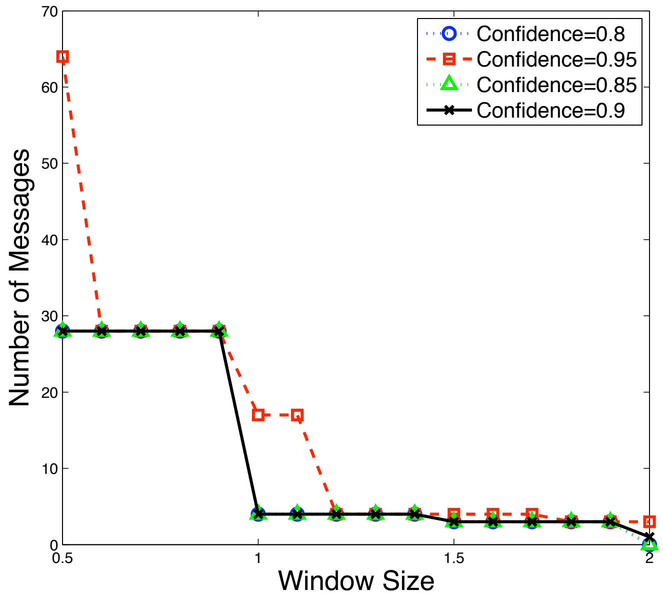


Figure 5.9. Query experiments on an in-network summary created using the set of window sizes [0.5 1 1.5 2].

will be more expensive to transmit and update. The size of each model is represented by the parameter k (for SGMs $k = 1$), and the amount of space that each model uses is proportional to k.

We compare 3 different types of models against SGMs:

k-mixture: A k-size mixture is maintained instead of a size 1 mixture. The compression of a l-size mixture to size k is done by clustering the l distributions of the original mixture into k sets using a modified k-means algorithm. Then each set is compressed to a SGM as described in Section 5.3.1.

Virtual nodes: As in k-mixtures, an l-size mixture is divided into k sets and a SGM is computed for each one. The difference is that in the previous approach, a single k-mixture represents all of the nodes in the subtree, whereas here each separate SGM represents only that portion of the nodes used to construct it. This is equivalent to "splitting" each node into k virtual nodes and using simple SGMs. Note that this approach requires some extra bookkeeping space to record the groupings of sensor nodes into the virtual nodes.

SGMs on multiple windows: The extra space is used to maintain additional SGMs for different window sizes. Depending on the query window the appropriate model is used at every node.

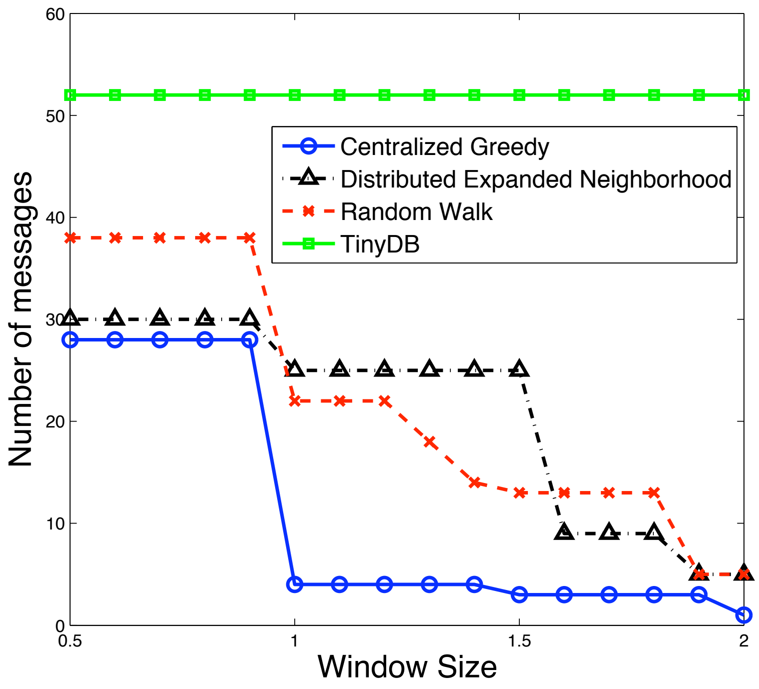


Figure 5.10. Query experiments on trees constructed by different clustering algorithms

The generalized models described are evaluated against simple SGM compression on a tree built for a workload of window sizes $[0.5, 1, 1.5, 2]$ and confidence of 0.9. The models are then tested on a query workload of a finer range (Figure 5.11). The results shown are for enriched models of size $k = 4$. Experiments with larger k were also performed, with no change in the results.

An initially surprising observation is that enriched models demonstrate minimal to no performance gains. Specifically, k-size mixtures are not any better than a size 1 mixture, and the same goes for virtual nodes. Even though this may seem counterintuitive, it is justified by the criterion used during tree construction. Nodes are divided into clusters ensuring that the mass in the interval $[c - w, c + w]$ for each cluster, where c the center of the cluster, is enough to satisfy the confidence that the tree was designed for. Also, the design of SGMs preserves this mass in the resulting compressed model. Therefore, a cluster designed on that criterion can be sufficiently represented by a SGM, making more elaborate models unnecessary for queries of equal or less confidence than the tree was designed for. We observe some minimal gains on queries of higher confidence – we can now receive some benefit from the additional information – but still the gains are not significant. These results are actually evidence of the quality of our tree construction method. Queries have requirements for normal error bounds and thus a normal distribution is the appropriate model when the underlying nodes are clustered based on its properties.

The third design, SGMs for multiple windows, shows some performance gains, but note that these are not due to modeling improvements, i.e. better representation of the underlying data, but better tuning to the design workload. A SGM hierarchy attempts to minimize the average communication cost for the design workload. The benefit in multi-window SGMs arises from keeping models of several window sizes on higher levels, thus giving the opportunity to queries to terminate higher in the tree than they otherwise would.

This observation, that keeping models of multiple window sizes can improve performance, may be an indication that assignment of window sizes across tree levels may also make a difference. We therefore want to evaluate our intuitive decision to assign window sizes to tree levels in decreasing order from root to leaves. We exhaustively enumerate all the possible assignments of window sizes from the design workload to different tree levels, and we compute their average communication cost on queries from that workload. As Figure 5.12 shows, the intuitive choice of decreasing window sizes along the tree levels is also the right one. The experiment was repeated with several different design workloads, with the same result.

Our analysis demonstrates that a SGM is sufficient for our data representation, making in-network summaries a very simple and effective approach for modeling sensornet data.

5.6 Parameter Sensitivity

In this section we evaluate the sensitivity of our tree construction algorithms to different parameters using simulation experiments. As with all the previous experiments, these are also performed over real temperature data from the Intel Berkeley Lab deployment.

First we examine the effects of the confidence parameter in tree construction. Two trees built on different confidence limits, 0.9 and 0.95, are evaluated over the same query workload of confidence 0.95. The results, shown in Figure 5.13, demonstrate that the choice of confidence for tree construction does not have a big impact on performance.

In our second set of experiments we want to explore the effects of the choice of design workload on communication performance during query time. Query workload W is evaluated over a tree T built with W as the design workload, and another one T' built with

$W' \subset W$ (Fig. 5.14). Interestingly enough T is not always the winner. This is because T is forced to have more levels than T', and for some window sizes (e.g. 0.5), it makes queries traverse more levels. As a result, including the full workload in the tree design is not necessarily the best choice. It would be interesting to explore how to optimize the window range for the tree design, and incorporating that range as a parameter in the optimization may require revisiting our tree construction heuristic.

In our final performance experiments, we evaluate the performance of in-network summaries over time. Data is taken for 48 hours, starting around midday of the first day. A summary hierarchy is built based on data from the first hour, and a workload W of window sizes $[0.5, 1, 1.5, 2]$ and confidence 0.9. When models at the leaves change due to temperature variations throughout the day, those get propagated up the tree either at query time, or with a background process. Models then get updated at various levels, but the structure of the tree remains the same throughout the experiment.

The hierarchy is tested using workload W every hour over a period of 48 hours, and compared with the performance of a tree that is completely rebuilt every hour (full restructuring). The results are given in Fig. 5.15.

We observe that even with full reconstruction we get bad performance when variance of the underlying data is high, as happens around timesteps 20 and 40. In the case of model updates without restructuring of the tree, even though for large windows the results are reasonable, for small windows they are very disappointing. Note however that performance deteriorates at different times for different windows. From the experimental results, the higher levels of the hierarchy corresponding to larger windows remain consistent until about 40 hours later, mid-levels (window 1) seem to become unusable after about 6 hours, and the lowest levels (smallest window) seem to not be at all reusable.

This behavior can be addressed with *escalated* restructuring. Different levels of the tree get restructured with different frequencies: clusters corresponding to small windows will get reclustered more frequently, whereas those of larger window size much less often. In Figure 5.16 escalated restructuring is performed every hour, 6 hours and 24 hours, for clusters of windows 0.5, 1, and 1.5 respectively. The choice of restructuring frequencies at this point is ad hoc and based on observations of the behavior in Fig. 5.15, but the purpose of this experiment is to demonstrate that escalated restructuring provides big performance benefits. An interesting topic for future work would be to automatically derive the appropriate frequencies or determine on the fly when reclustering is necessary.

In-network summaries with escalated restructuring have performance comparable to the best result. After time 42 however, the model deteriorates, because coincidentally the size 1 clusters get updated on a moment of high variance of the underlying data, and thus the resulting clusters are of low quality. A simple fix would be to detect high variance in the data, and avoid restructuring during those times.

Using escalated reclustering, in-network summaries are able to follow changes of the data that happen at a relatively slow rate. This makes our scheme suitable for various monitoring applications that deal with slow changing phenomena, like environment observation and forecasting systems, various types of habitat monitoring applications, landslide detections, water quality monitoring. It is not suitable for emergency response situations that involve sudden changes in the underlying data, like fire alert systems. In order to deal with cases that require some type of outlier detection, in future work, in-network summaries could potentially be augmented with a push scheme that locally detects sudden data changes and reports them to higher levels.

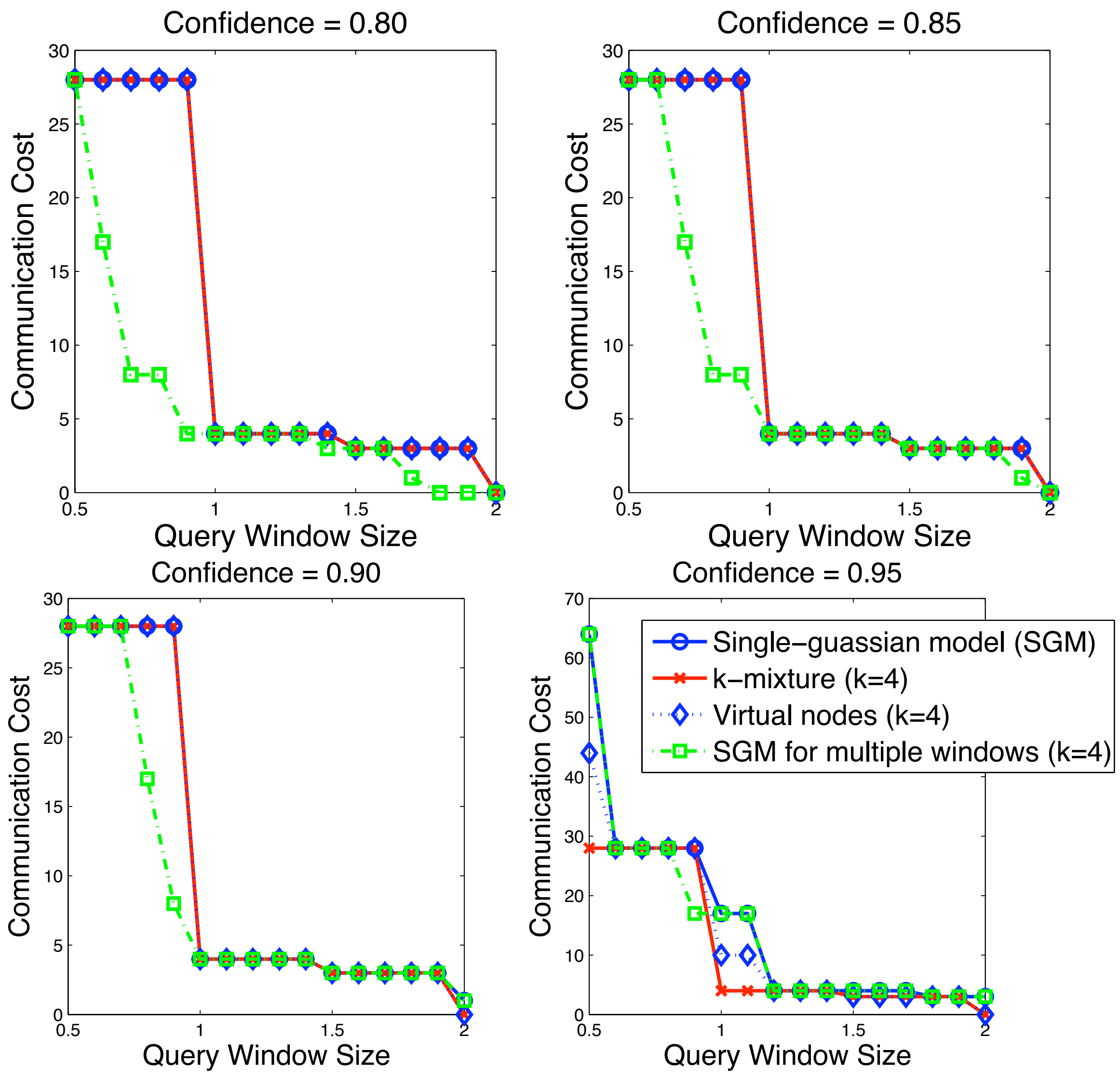


Figure 5.11. Evaluation of SGMs and enriched models

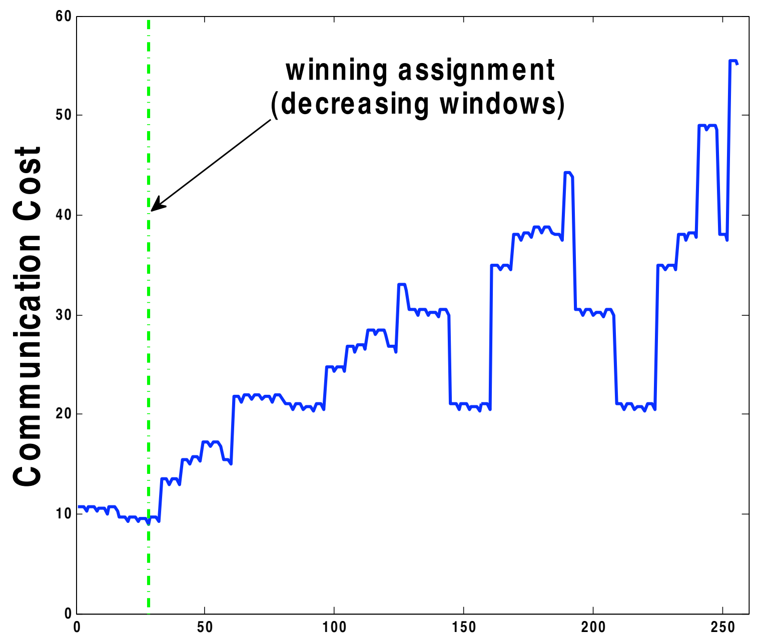


Figure 5.12. Evaluation of window assignments across tree levels

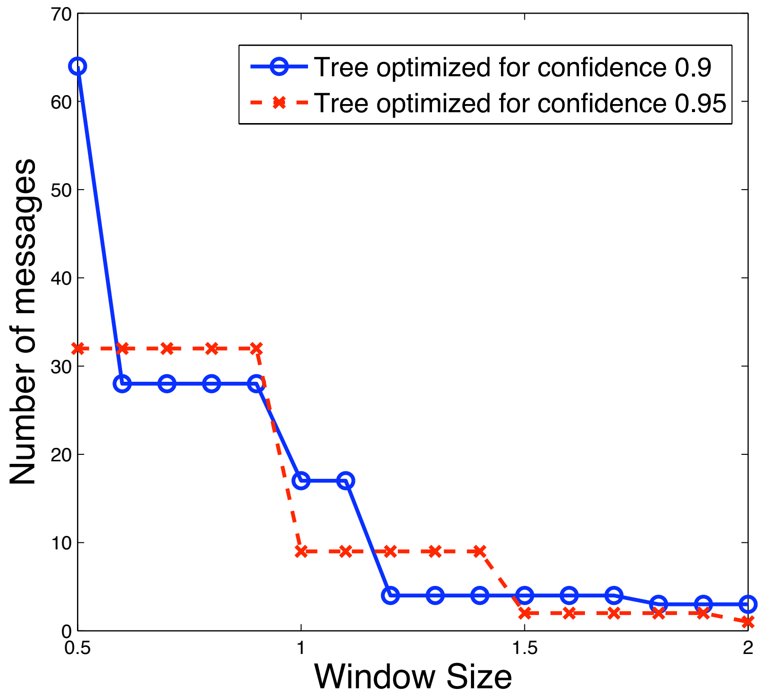


Figure 5.13. Comparison of hierarchies built on different confidence. The query workload is of confidence 0.95.

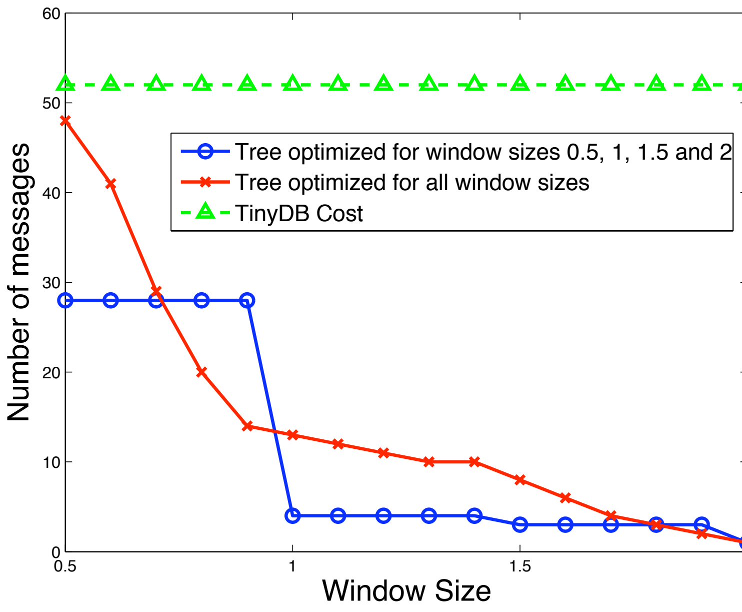


Figure 5.14. Comparing tree construction with a few vs a broader range of windows

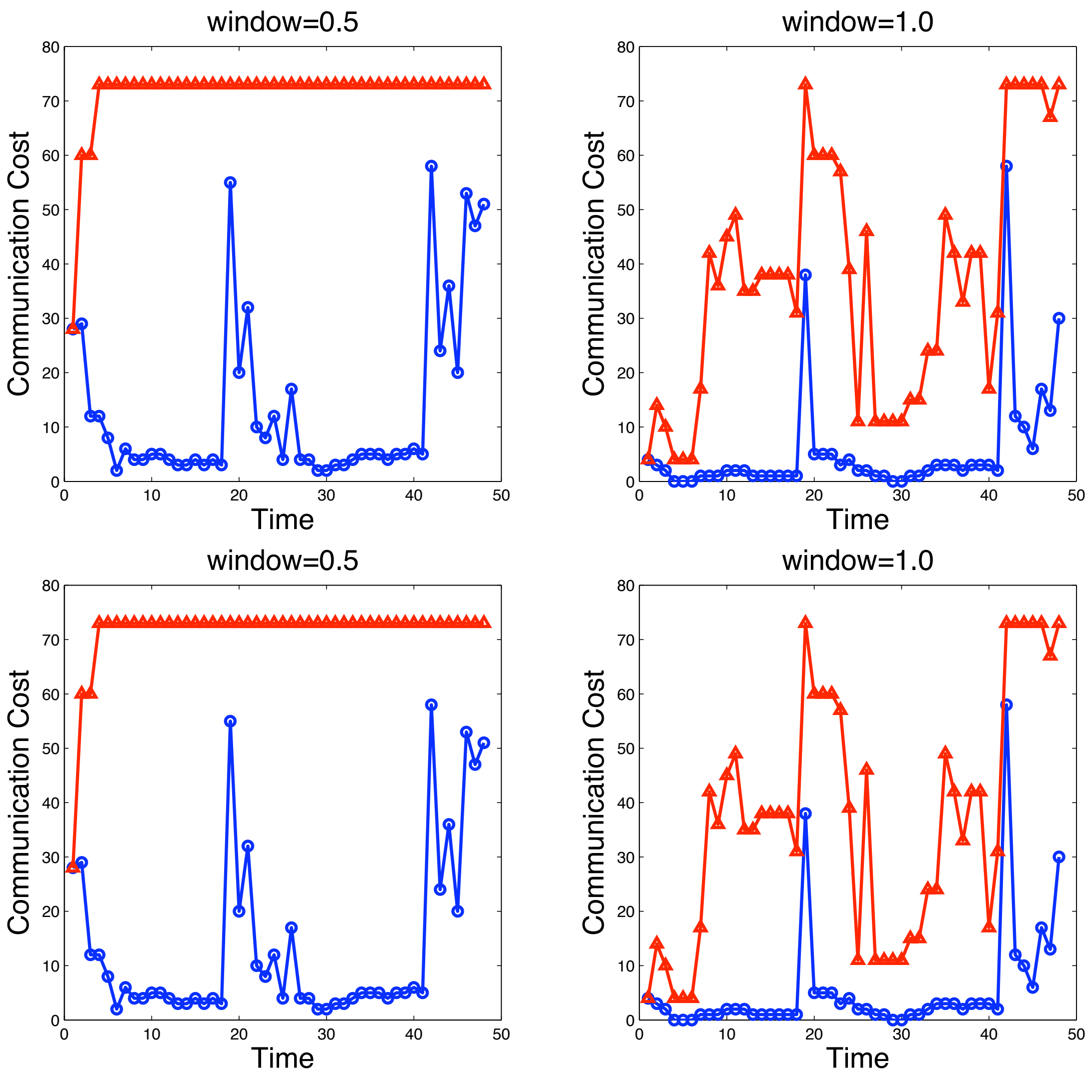


Figure 5.15. Time progression of in-network summaries with model updates.

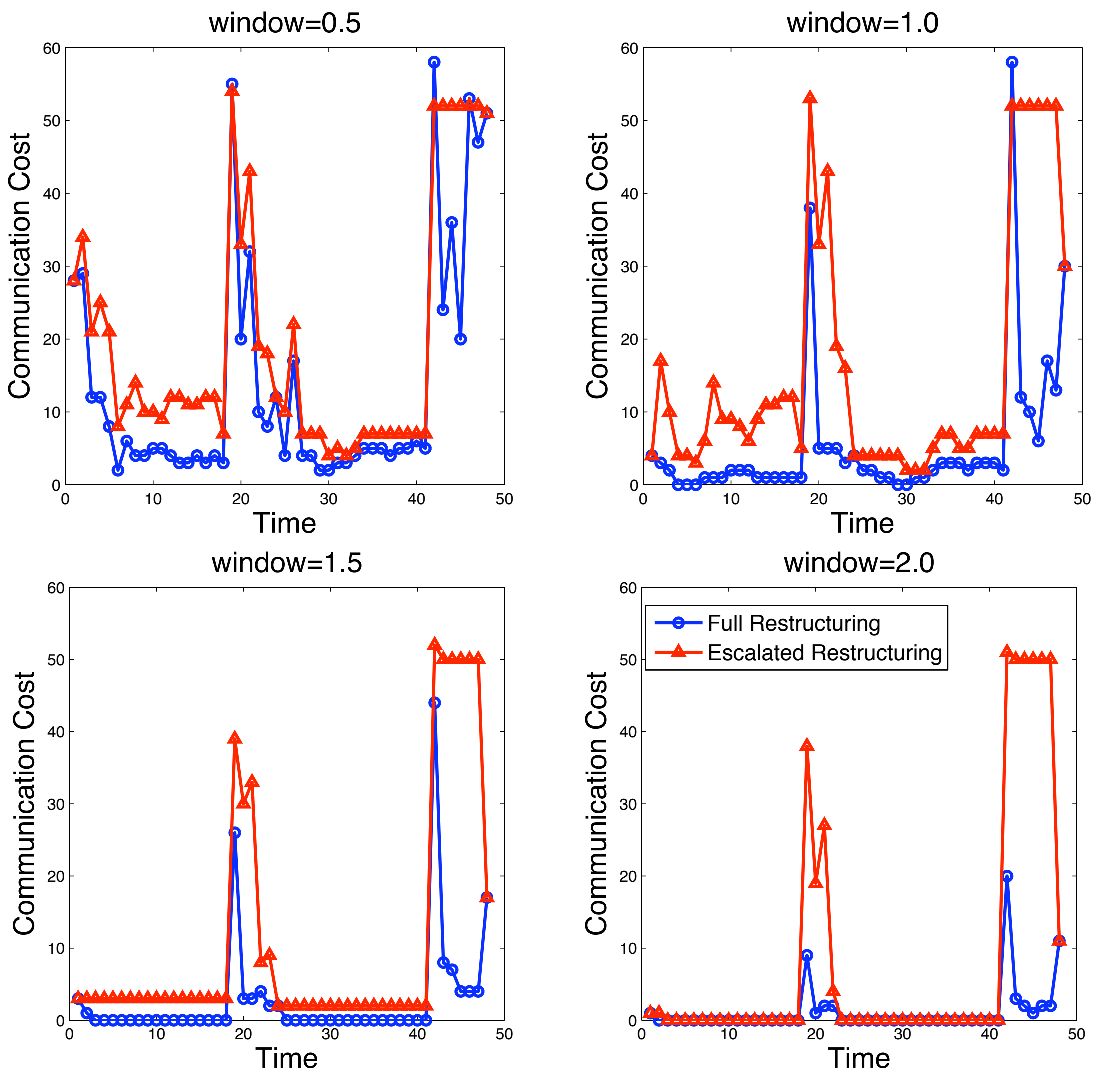


Figure 5.16. Time progression of in-network summaries with model updates and escalated restructuring.

Chapter 6

Distributed Estimators

The focus of this thesis is the optimization of sensor network queries in terms of communication cost: given a query, what is the observation plan that can provide a satisfactory answer with the least number of messages. In previous chapters we used modeling techniques to reduce the number of measurements that need to be collected, and we developed planning algorithms that seek the optimal observation plan based on the stored models and prior knowledge of the network connectivity. Chapter 5 exploited model-driven techniques in a distributed setting. Centralized techniques can provide more holistic and overall optimal solutions, but are harder to maintain, scale, and are harder to recover from failures and are more sensitive to changes of the network topology.

Up to now we have been focusing on value queries, i.e. queries requesting values at different sensor locations. In this chapter we will address *aggregate* queries, i.e. queries that request a certain type of summary information over a group of sensor nodes. Aggregate queries are very common in a sensornet setting due to the nature of the data. Sensors commonly measure physical phenomena, where an aggregate query can often provide sufficient information for the application (e.g. what is the average/maximum temperature in the room).

Aggregates are by nature data summaries, making inaccuracies in the estimates of single node values less critical, and their differences from `SELECT *` queries require us to explore new approaches for their optimization. In Chapter 5, we presented a technique to summarize information along an in-network hierarchy to make evaluation of `SELECT *` queries more efficient. In this chapter we examine types of summary information suitable for aggregate queries. Similar to our in-network summaries, we maintain a hierarchical approach, this time tuned for aggregation. We first discuss *spatial interest queries*, which define regions over which aggregates are to be computed. Our scheme generalizes Data Cubes [37] to represent area aggregates at different resolutions over the network, and employs them over a grid overlay. In Section 6.2 we study the optimization of spatial aggregate queries over a grid of deterministic values, and discuss cube construction and recovery from failures. Finally, Section 6.3 discusses the generalization to probabilistic values over the grid locations.

6.1 Spatial Interest Queries

In Chapter 3 we focused on selective data gathering, which could result either from queries over part of the network, or from selectivity imposed from model-driven techniques. When this selectivity is defined by a geographical area, then we have *spatial interest queries*. Spatial interest queries basically define geographical areas over which the aggregate is defined. These are not to be confused with queries over geometry data: whether a region is contained in another, or whether two rectangles intersect. Spatial interest queries still need to resolve the containment of sensor nodes in the specified area of interest, in order to decide which sensors should be queried to produce a result, but the queries are defined over the measurement data of sensors, as opposed to spatial data.

Spatial interest queries are a common occurrence in sensor networks, as the application may sometimes need to monitor only part of the phenomenon covered by the deployment. They are basically queries which define a spatial region as a selection criterion, and query results are computed based on data from nodes that are located within that region. They are used to answer questions like: "What is the average temperature in region R". Unlike traditional attribute queries, spatial queries require that the sensor network can process location information as well. In this chapter we will study in-network summarization schemes that support spatial aggregate queries, i.e. queries that request aggregate information over a region of interest.

6.1.1 Related Work

Spatial query processing has been extensively studied in centralized systems. The R-tree [40], and its variants ([11], [71]), is one family of index structures for spatial data. In the R-tree, each spatial data object is represented by a Minimum Bounding Rectangle containing a pointer to the object in the database. Non-leaf nodes store a MBR that contains the MBRs of all the children nodes. A query traverses the R-tree using "containment" and "overlap" checks to appropriately navigate through the structure. Spatial indexing is discussed more extensively in [39].

Because of resource limitations in sensor networks building centralized indexes is often not practical. [27] proposes a peer-tree, a distributed R-tree using peer-to-peer techniques, partitioning the sensor network into hierarchical, rectangle shaped clusters. The techniques include joins and splits of clusters, and the authors show how to use the structure to answer nearest neighbor queries. The use of quad-trees in such setting is also natural and [26] uses distributed quad-trees to support spatial querying.

SPIX [76], is a distributed spatial index which uses Minimum Bounding Areas in an R-tree like fashion. The spatial query processor on each sensor uses SPIX to bound the branches that do not lead to results, find a path to sensors that do have results to report, as well as aggregate data.

In the database community, a number of distributed and push-down based approaches have been proposed for aggregation ([72], [83]). However these assume a well connected, low-loss topology that is often unrealistic in sensor networks. TAG ([56], [58]) performs in-network aggregation to reduce the amount of data that needs to be send over the network. TAG provides a simple declarative interface for aggregation, and it distributes and executes aggregation operators in the network, computing aggregates through data flow. Aggregation in adversarial settings is examined in [36].

In this work we present an in-network summarization scheme that supports spatial aggregate queries. The goal is to minimize the number of locations that need to be queried in order to retrieve the proper aggregate, differentiating our approach from the other aggregation techniques. We follow a data-cube like approach [37], which uses simple aggregates for summarization, like *prefix-sum* [45].

Aggregate Types

Aggregate functions can be classified into different categories based on their state requirements, tolerance of loss, duplicate sensitivity, and monotonicity ([37], [56]). Different types of aggregate functions require different types of summarization, as they differ in the way that partial state aggregates can be used to compute a global aggregate.

Distributive: The partial state is simply the aggregate, over the corresponding partition of the data. Examples are `MIN, MAX, SUM` aggregates.

Algebraic: Partial state is of constant size, even though they do not themselves represent the aggregate of the corresponding data (e.g. `AVG`).

Holistic: Partial state is proportional to the size of the data, and therefore in this case partial aggregation is not useful (e.g. median).

Unique: These are similar to holistic aggregates, but the size of the state is proportional to the number of distinct values in the partition (e.g. distinct count).

Content-Sensitive: The size of the partial states depends on some property of the data (e.g. histograms).

In this work we will focus on distributive and algebraic aggregates. In the discussion we will mainly talk about sums and averages, but the approach is straightforward to generalize to arbitrary algebraic functions.

6.1.2 Multiresolution Cubes

Our goal is to build a framework to efficiently respond to aggregate queries with spatial constraints. Such queries are common in many applications, and are of the form: "`SELECT avg(temperature) WHERE nodeID inRegion [(2,3),(5,9)]`".

Regions of interest can be defined as a set of points on the plane, and we can easily generalize to 3 dimensions. In the simplest case, the region of interest is a rectangle that can be defined by just 2 corners, for example upper left and lower right. Regions can be of arbitrary shape, but those can always be split into simpler rectangles. The corner points of the interest areas can be given as coordinates in the euclidean space. Without loss of generality, we can assume a grid of sensing locations and regions of interest defined over this grid. The assumption of the grid makes it easier to define the queries' area of interest, without restricting the application domain to grid deployments. The grid locations do not need to correspond to actual sensor locations, as the grid can be an overlay over the actual network topology (more on that in Section 6.3). For now we will make the simplifying assumption that the sensors are placed on a grid.

Queries define an area of interest over the grid, specified by the locations of corner points, and can be of arbitrary rectilinear shape. An example is shown in Figure 6.1.

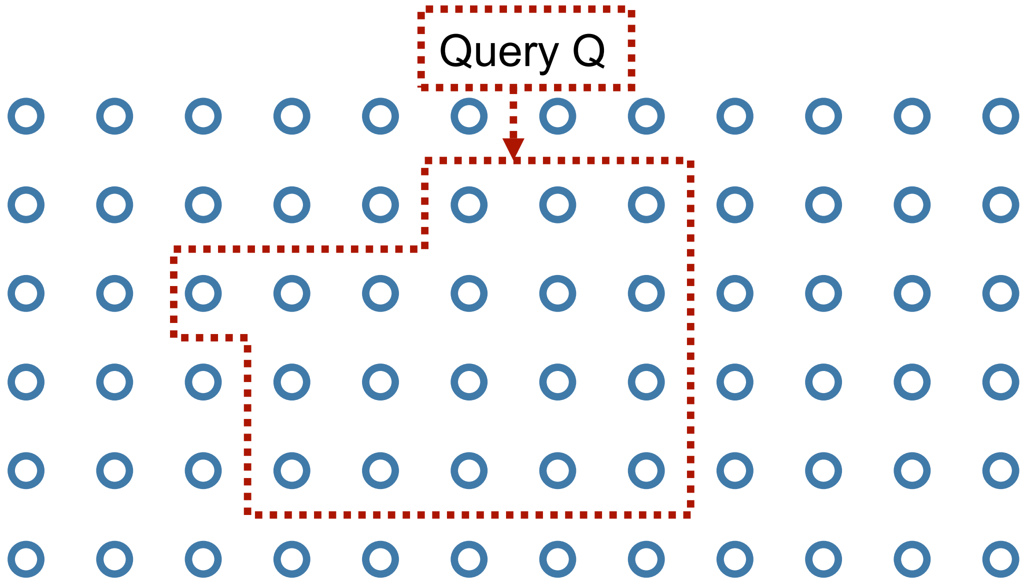


Figure 6.1. Spatial Queries can be over arbitrary areas of the grid.

Our approach introduces an in-network *multiresolution data cube* scheme, which allows for efficient computation of area aggregates. The cube hierarchies can be constructed in a distributed fashion, and in case of failures, recovery can also be performed with in-network decisions. The grid provides a natural setting for constructing cube summaries, as those can be computed along the grid directions, as in traditional datacubes. Without loss of generality we can assume that the top left corner of the grid is location $(0, 0)$, with coordinates increasing from left to right and top to bottom, and that we maintain sum information along the increasing directions.

Computing and maintaining a cube structure over the whole network introduces a lot of latency as values need to be propagated from one end of the network to the other.

In order for the scheme to scale we introduce *multiresolution cubes*. The grid is divided into rectangular cells of equal size and each one is treated as a separate data cube, and the summary data computed separately. The cells are not overlapping and they cover the whole grid. The final summary information for each cube is stored at one of the cell locations (without loss of generality we pick the lower right corner). The cells that tile the grid on this level can be grouped together in bigger tiles/cells. The new summary information at the level of the bigger cells can be computed using the cells at the lower level as entities.

This process defines a hierarchy of increasingly bigger cube-cells, eventually forming a structure resembling a quad-tree. A depiction of this process can be seen in Figure 6.2.

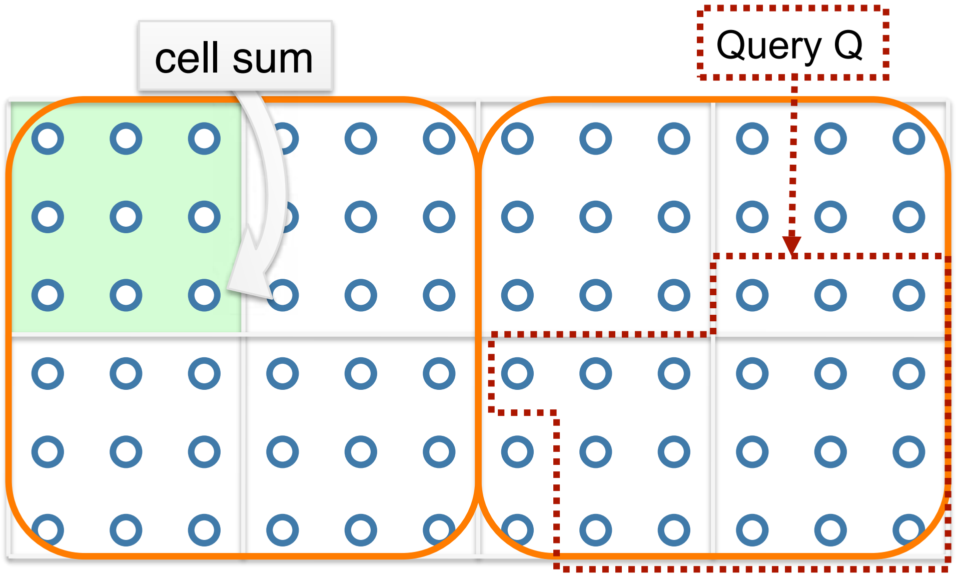


Figure 6.2. Division of the grid into cells and forming a multiresolution cube with increasingly bigger cells. Queries can span cells of different granularities.

The cell division in the hierarchy follows the following rules:

- Cells of the same level do not intersect.
- If cell A of level i, and cell B of level j, where $i > j$ (meaning that level i is higher than level j) intersect, then $B \subset A$.

Mapping query regions to cells

The hierarchy cells hold summary information that can be used to construct query answers. An arbitrary query region can be decomposed into smaller rectangles based on the cell hierarchy. Then the region of interest becomes a collection of non-overlapping cells, the summary data of which can be used to compute the aggregate over the whole region. Since cells of the same hierarchy level are not overlapping, the query area can be greedily split into cells in an optimal way, minimizing the number of cells that comprise it.

Algorithm 11 Greedy Division (pseudocode)

```
1: repeat
2:     Select an convex corner $c$ of the region of interest.
3:     Find largest hierarchy cell contained in the region, that shares the same corner $c$.
4:     Extract cell from region.
5: until Region is empty
```

Algorithm 11 divides a query region into the minimum number of hierarchy cells. In the algorithm description, a *convex* corner is a corner of the region of interest in the immediate area of which the region is convex. If a corner is not convex, it is *concave.*

Lemma 5 *The greedy division of query regions into hierarchy cells produces a set of cells with minimum cardinality.*

Proof: Since cells of the same level do not overlap, every choice of cell with the greedy algorithm elects a maximal cell – no bigger cell that contains the same area could have been selected. Also, a greedy choice of cell does not eliminate solutions that use part of the chosen region, because of the no-overlap rule, and therefore the greedy division algorithm splits the query region into the minimum number of cells. ■

6.2 Deterministic Analysis

Multiresolution cubes form a structure in the network that could use different types of summarization data. As a first step we will assume deterministic data, a fixed value at each grid location, and simple sums stored at the lower right corner node of every cell level, as shown in Figure 6.2. For this discussion, we also assume without loss of generality that queries request `SUM` aggregates. The approach can easily accommodate other distributive aggregate types.

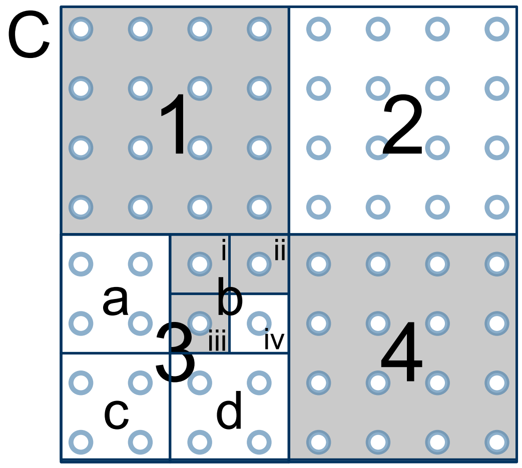


Figure 6.3. The grey area depicts the area of interest of a query over the grid. It is comprised by cells $G = \{1, 4, i, ii, iii\}$.

Figure 6.3 shows an example of a query Q, defining an area of interest (G) over the grid. The region of interest is depicted by the grey shaded areas. Area G is split into hierarchy cells using the Greedy Division Algorithm (Algorithm 11). One possible approach to answering the query is to retrieve the sums of the cells in G, i.e. cells 1, 4, i, ii, and iii, and construct from them the aggregate value in the interest area $V(G) = V(1)+V(4)+V(i)+V(ii)+V(iii)$, where $V(A)$ refers to the aggregate value (sum) stored for cell A. The solution is however not unique. We can also compute $V(G)$ as $V(C) - V(2) - V(a) - V(c) - V(d) - V(iv)$, and there are numerous other possibilities.

We define the optimal query plan for an aggregate query with spatial constraints, as the plan that requires the retrieval of the least number of elements from the grid, and the goal is to select this optimal plan from the solution space.

6.2.1 Query Optimization

In this section we will show how to compute the optimal plan for a spatial aggregate query in polynomial time, using a simple transformation to a max-flow min-cut problem. The example query displayed in Figure 6.3 can also be represented as a tree hierarchy, corresponding to the hierarchy of cells of the multiresolution cube. This is depicted in the left side of Figure 6.4. The direction of the edges represents the transfer of "mass" (or sum information) from a node to its parent. Each node only has one outgoing edge, the mass of which is equal to the sum of the masses of the incoming edges. This "preservation" of mass means that the outgoing mass of a node is equal to the sum of the outgoing masses of its children.

The hierarchy can be transformed as shown in the right part of Figure 6.4. The structure is exactly the same, but the area of interest has been separated from the rest of the cells, by placing the shaded *grey nodes* on the left side of the graph and the unshaded *white nodes* on the right side. Those nodes that have both grey and white descendants are left in the middle, "undecided". The white (right) side also contains a node denoted as "root", representing all the ancestors of C.

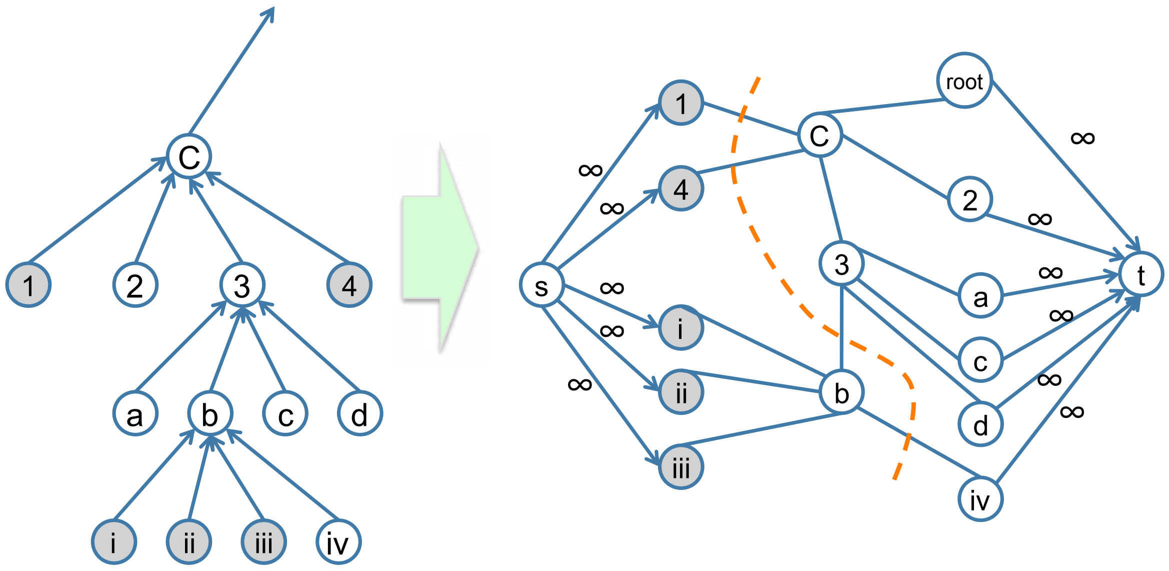


Figure 6.4. Transformation into a max flow problem. The minimum cut is the best solution: V(1)+V(4)+V(b)-V(iv).

The transformation of the cube hierarchy into a max-flow graph is completed by:

- Removing directionality of edges, and assuming capacity of 1 for all existing edges
- Adding source node s on the grey side and connecting it with infinite capacity edges to each node in the grey area

- Adding target node t on the white side and connecting it with infinite capacity edges from each node in the white area

Solving the max-flow min-cut problem on the *transformation graph* produces the optimal observation plan for the query.

Theorem 10 *The minimum s-t cut in the flow graph is equivalent to the optimal query plan.*

Proof: Due to the preservation of mass it is easy to see that the mass [1] along the grey border is equal to the mass along the white border.

Every s-t cut (which does not involve infinite capacity edges) simply "chooses sides" (grey or white) for the intermediate nodes. Also, any s-t cut can be produced from another s-t cut by appropriately changing sides to the intermediate nodes.

The mass along the grey border is the mass we are interested in (mass of the grey area M_G). We start from cut C_G which is the cut along the grey border. Assume C_m the minimum cut which differs from C_G in the set of intermediate nodes $I = \{i_1, \ldots, i_k\}$ which C_m assigns to the grey side. C_G can be transformed into G_m by switching sides to the set I one by one.

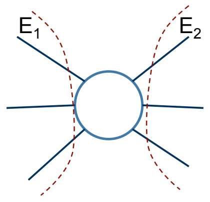


Figure 6.5. $Mass_{E_1} = Mass_{E_2}$

Note that by "flipping" sides for one node, the mass is preserved, because for every node the sum of all mass (incoming and outgoing) is zero. During the switch, edges E_1 are exchanged for edges E_2. But $Mass_{E_1} = Mass_{E_2}$ [2]. Therefore, $Mass_{C_m} = Mass_{C_G}$.

Also, because C_m is the cut with the minimum number of edges, it's the optimal query plan. ■

[1]We use the term mass corresponding to the aggregate sums, and not the capacity of edges in the flow graph

[2]Note that values of mass will be added or subtracted based on the direction of the edges in the cut: incoming or outgoing.

This gives us a polynomial algorithm for computing the best query plan over a cube hierarchy. We can use the Ford-Fulkerson algorithm to solve the corresponding max flow problem, which is of complexity $O(E * maxFlow)$. In our case since we have unit capacity edges, the *maxFlow* parameter is bounded by $O(n)$.

6.2.2 Multiple Queries

In this Section we will examine the case of multiple queries running at the same time in the system. Optimizing them individually as discussed in the previous section produces the most efficient result for each one individually, but not for all of them together. To understand this, consider the following example:

Assume the area divisions as shown in Figure 6.3. Query 1 (Q_1) is the one depicted in the figure: $G^{Q_1} = \{1, 4, i, ii, iii\}$. Also assume query Q_2 for which $Q^{Q_2} = \{1, 4, i, ii\}$. Obviously the two queries share a large common area. Optimizing for Q_1 produces the solution $V(1) + V(4) + V(b) - V(iv)$ using the max-flow transformation described in the previous section. Optimizing for Q_2 produces $V(1) + V(4) + V(i) + V(ii)$. This means that for both queries we end up requesting 6 elements in total.

This is obviously not the best solution as we can compute both query results by retrieving just 5 elements: $\{1, 4, i, ii, iii\}$.

We solve the problem of multiple query optimization by creating a *combined transformation graph* for all the queries that are running simultaneously. The individual transformation graphs for Q_1 and Q_2 are given by figures 6.4 and 6.6 respectively.

The two transformation graphs can be merged into a combined transformation graph using the following simple rules:

- If a node is of the same color (grey, white or undetermined) in both graphs, then it remains of the same color in the combined graph.
- If a node is of different colors in the two graphs, then 2 “images” of the node are added to the combined graph, one with each coloring.
- If a node exists in one graph but not the other, it will appear in the combined graph based on its properties in the graph it appears in.
- All edges of the original graphs are replicated in the combined graph.

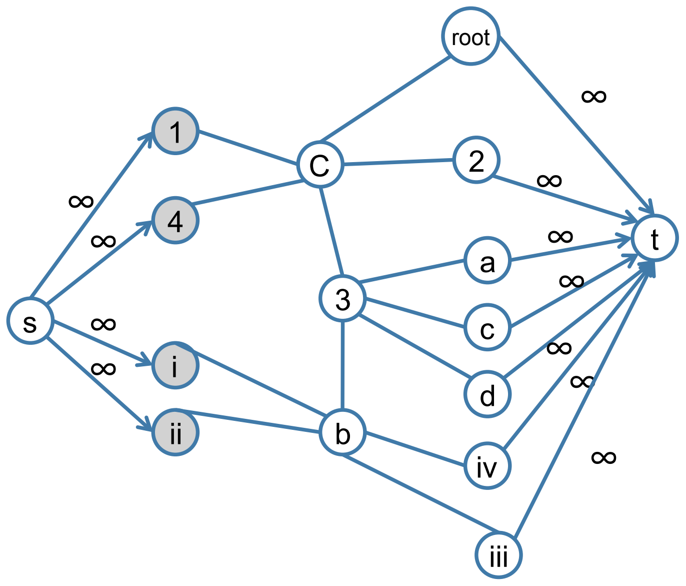


Figure 6.6. Max-flow transformation graph for example query Q_2.

The combined graph for our example is depicted in Figure 6.7. Solving the max-flow problem in the combined graph will produce the desired solution. Also from the same graph it is now easy to see that $\{1, 4, b, iii, iv\}$ would be an equivalently good solution, also of 5 elements.

The combined graph contains all the information of the original graphs. Any cut in the combined graph defines a cut in the original graphs as well, and therefore a cut in the combined graph provides solutions to all queries that created it.

6.2.3 Prefix-Sum Hierarchies

Up to now we have based our analysis on a simple cube scheme, where the only information kept at every cell was the total sum of elements in the cell. The summarization scheme however can be more elaborate, storing more information in each cell than just a single value.

A simple summation scheme that does that is the *prefix-sum* algorithm [45]. Prefix sum (PS) works on the grid by storing at location (i^*, j^*) the sum of the values from all locations where $i < i^*$ and $j < j^*$. Figure 6.8 displays an example of the prefix sum algorithm. The left matrix contains the individual data, representing the values at each location of the grid, and the right matrix shows the result of the prefix-summation.

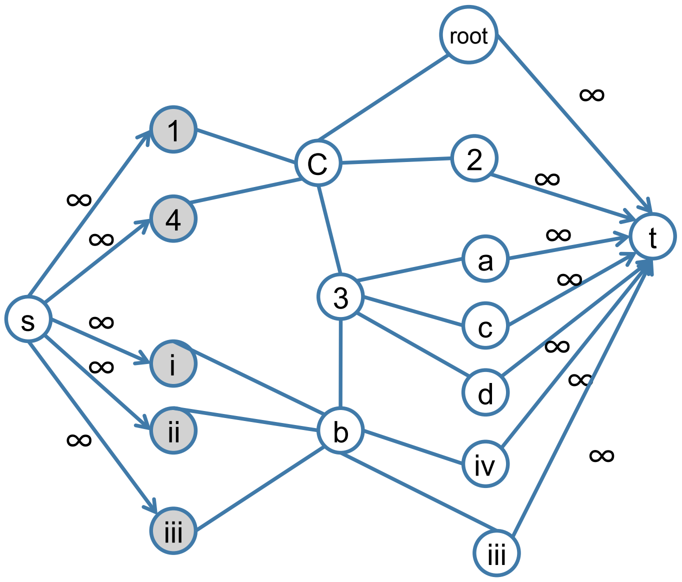


Figure 6.7. Combined Transformation Graph.

2	10	10	5	9	2
5	10	8	10	7	1
9	10	7	8	8	4
7	2	8	1	3	1
1	9	7	4	10	1
5	4	8	8	2	5

2	12	22	27	36	38
7	27	45	60	76	79
16	46	71	94	118	125
23	55	88	112	139	147
24	65	105	133	170	179
29	74	122	158	197	211

Figure 6.8. Example of Prefix-Sum

The difference with the previous summation scheme is that PS values are stored in all nodes instead of just one node per cell. Again we can build multiresolution hierarchies, by splitting the grid into cells at the lowest level, performing PS at each one, and reiterating at the next level with each cell as a new base element. The value of each cell used at the higher level as a base element would again be the lower right corner value, where the prefix-sum would store the total sum of the cell. As would also be the case in the simple sums scheme, the lower right element of the grid would end up with the total sum of the whole grid, as the corner node of the root cell of the multiresolution cube.

One of the advantages of prefix sum is that it provides finer granularity at the aggregate computation for a query. Assume a query area that is smaller than the lowest level cell, or only contains part of a cell. The cube would not have been useful in the simple sum case, but if it was constructed using prefix-sum, it can still be used for optimization.

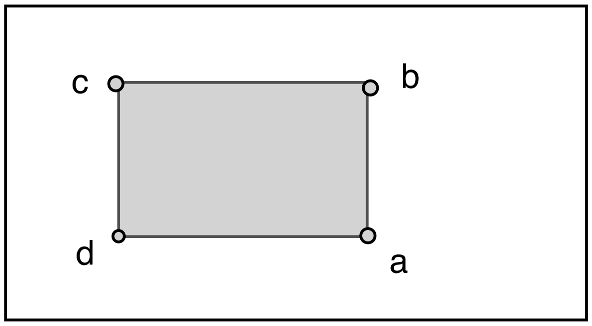


Figure 6.9. The sum in the rectangle is $(a - b + c - d)$.

For every rectangular region the aggregate (sum) in the region can be calculated by exactly 4 numbers, the PS values at the corners of the rectangle (Figure 6.9). The sum in the rectangular region is given as: lower right corner + upper left corner - upper right corner - lower left corner. This is a straightforward observation from the definition of prefix-sum.

The aggregate of an arbitrary shaped region can be calculated by splitting the region into simpler rectangles. However, even though there are many possibilities of splits, at the end the number of elements that we need to retrieve is constant and depends on the shape of the region.

Lemma 6 *The total number of values needed to calculate the sum in an arbitrary region is the same as the number of its corners.*

Proof: Dividing the region into rectangles uses existing corners and creates some new ones. New corners, i.e. points that were not corners in the original interest region but were created due to a division, will participate in the summation an even number of times, alternating between summation and addition, canceling each other out. This is made clear by Figure 6.10. A "created" corner will have to participate into the sum an even number of times due to being a corner to two or four adjacent "created" rectangles. But as we can see from the right part of Figure 6.10, which is a simple depiction of the way sums are computed for a simple rectangle, adjacent rectangles make the corner introduce its sum with alternating signs: positive and negative, depending on whether it is the lower right, lower left, upper right or upper left corner of the current rectangle.

The actual corners (the points that were corners in the original region) will participate to the summation once or thrice, ending up in the total either added once or subtracted once. Hence the result of the above lemma. ∎

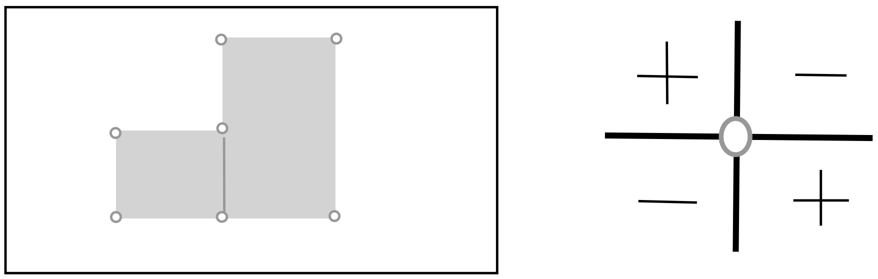


Figure 6.10. Each corner gets added or subtracted depending on its position relatively to the current rectangle.

Query Answering on a PS cube

The technique we presented to construct query plans in a simple sum hierarchy can also be used in PS hierarchies as well, as the summaries stored in a prefix-sum cube are a superset of the simple summation. The difference is that the PS scheme contains extra information, within each cell, which can be used to optimize for aggregates over regions of interest when they are divided into cells of the same granularity. In the case of the 3×3

cube shown in Figure 6.11, the simple sum algorithm would give us access to 10 elements to work with:

```
[1], [2], [3], [4], [5], [6], [7], [8], [9], [1,2,3,4,5,6,7,8,9]
```

where $[s_1, \ldots, s_i]$ represents the sum $s_1 + \ldots s_i$.

On the other hand, in a $k \times k$ cell of a PS cube, we store $2k^2 - 1$ different sums. In the case of our example, those are:

```
[1], [2], [3], [4], [5], [6], [7], [8], [9], [1,2], [1,2,3], [1,4],
[1,2,4,5], [1,2,3,4,5,6], [1,4,7], [1,2,4,5,7,8], [1,2,3,4,5,6,7,8,9]
```

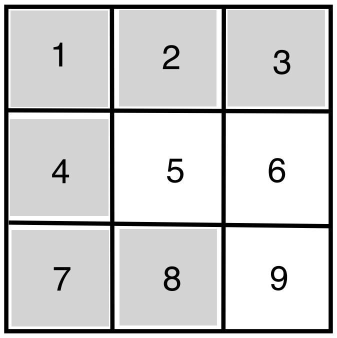


Figure 6.11. Depiction of cells with query (grey).

These are divided into 3 groups based on whether they belong to the region of interest or not: grey (S_g), white (S_w) and undetermined (S_u). S_g contains elements that all belong to the region of interest, S_w contains elements that do not intersect the region of interest, and S_u contains elements that partially intersect the region of interest. For our example these sets are:

S_g:

```
[1], [2], [3], [4], [7], [8], [1,2], [1,2,3], [1,4], [1,4,7]
```

S_w:

```
[5], [6], [9]
```

S_u:

```
[1,2,4,5], [1,2,3,4,5,6], [1,2,4,5,7,8], [1,2,3,4,5,6,7,8,9]
```

We transform undetermined set S_u into "re-colored" set S_c, by appropriately subtracting from each element in S_u elements from S_w, so that remaining regions are grey.

S_c**:**

```
[1,2,4,5]-[5], [1,2,3,4,5,6]-[5]-[6], [1,2,4,5,7,8]-[5],
[1,2,3,4,5,6,7,8,9]-[5]-[6]-[9]
```

Each element in set S_g has weight 1, and the elements in S_c have weight determined by the number of elements used to produce them. In our example these costs would be 2, 3, 2 and 4 respectively.

We can now construct the best query plan on our prefix sum cube with dynamic programming, using Algorithm 12, where G is the grey area of interest, $S = S_g + S_c$ and $c = 0$:

Algorithm 12 PSQuery(G,S,c)

```
1: if $G = \emptyset$ or $S = \emptyset$ then
2:     return c
3: end if
4: for s in S do
5:     i=i+1
6:     G'=G-s
7:     S'={$s \in S$ s.t. s contained in G}
8:     c'=c+cost(s)
9:     A(i)=PSQuery(G',S',c')
10: end for
11: return min($A(i)$)
```

6.2.4 Building Multiresolution Cubes

When constructing a multiresolution cube, the first design choice is the size of the cells at each level, which corresponds to the hierarchy fanout. Starting from unit cells at the lowest level, the fanout parameter F determines how big the cells one level up will be (2×2, 3×3 etc). Smaller fanout creates more detailed hierarchies, with more levels, which results in higher storage costs. Bigger fanout results in coarser cells, and smaller hierarchies.

It is possible to have different fanout at different levels. If we keep the fanout constant across the hierarchy, then it is easy to compute the optimal value of F given a query workload. The query workload is defined as a set of regions of interest. Since we have a polynomial algorithm to compute the cost of a query over a specific hierarchy, we can use it to compute the cost of responding to the query workload for different values of the fanout. The hierarchy itself does not need to be constructed for this process. The algorithm will still be polynomial in the size of maximum fanout, which is bounded by the dimension of the largest cell that can fit in a query in our workload.

Distributed Construction

Construction of the cube should be done in a distributed fashion, and should not be communication intensive. In this section we will present a distributed algorithm for the cube construction, which requires only a single transmission by every node.

We will assume that nodes know their location on the grid, and that a single packet has enough space for h values, where h is the height of the hierarchy we want to construct. Every node will store up to h values, depending on how many levels of the hierarchy it's participating in.

Given a set of fanouts $\{F_i\}$ for the various hierarchy levels, a node can tell if it serves as a *junction* for level k, i.e. the node that stores the sum of the current level-k cell, if both its coordinates are divisible by $\prod_1^k F_i$. If a node is a junction for level k, it needs to forward its k-level sum to level $k+1$, but nothing to levels $> k+1$. A node adds its $k-1$ value, to the k-level message, thus adding to the k-level sum. Also, a node knows based on its coordinates if it lies at the border of a level region.

More specifically, the algorithm works as follows:

- A node with coordinates (x,y) expects messages from nodes (x,y-1), (x-1,y) and (x-1,y-1). If one or more of these nodes don't exist ((x,y) is at the edge of the grid), then (x,y) proceeds without those messages.
- A packet is of the following form: $P = [F_1 : value, F_2 : value, \ldots, F_h : value]$.

- F_0 is defined as the local value at every node.
- Every node will store $k+1$ values, where k is the level for which the node is a junction. By default all nodes are junctions for level 0.
- Upon receipt of the 3 packets P_a from (x,y-1), P_b from (x-1,y) and P_c from (x-1,y-1), it computes $P(F_i) = P_a(F_i) + P_b(F_i) - P_c(F_i)$.
- If x-1 is divisible by $\prod_1^k F_i$, then node (x,y) sets $P_b(F_k) = P_c(F_k) = 0$ before making the above computation. If y-1 is divisible by $\prod_1^k F_i$, then node (x,y) sets $P_a(F_k) = P_c(F_k) = 0$ before making the above computation.
- A k-level junction node stores level values up to level $k+1$, where for level i the value is $P(F_i) + local(F_{i-1})$.
- The node builds a new packet P with the $k+1$ values that it has stored. It also populates it with the values $P(F_{k+2}), \dots P(F_h)$ as computed upon receipt.

With this scheme, every node broadcasts one packet, and receives 3. Note that the described algorithm supports prefix sum hierarchies, but can be very easily transformed to the simple scheme by dropping the $k+1$ level values stored at the nodes.

6.2.5 Failures

From the algorithm for constructing the sum hierarchies presented in Section 6.2.4, it is obvious that values for most nodes can be reconstructed by simply querying 3 immediate neighbors.

The k-level value for a specific node is constructed as $local(F_i) = P_a(F_i) + P_b(F_i) - P_c(F_i) + local(F_{i-1})$. Therefore, any node that is used as an a,b or c-node in that equation can have its value easily reconstructed by querying the other 3 nodes in the local square.

The only problem arises with junction nodes. A junction node does propagate its value in the same fashion, but the receiving nodes do not store it but only forward it until it reaches the appropriate node for that hierarchy level. That means that if a node fails that serves as a junction for level i, then we need to travel distance equal to $3F_i$ (F_i the current fan-out) to retrieve the missing value.

Another alternative would be to store at every node up to level $k+2$ values instead of $k+1$. The construction scheme would not change, as that information is passed around the network anyways, but nodes would be required to store one extra value. That would allow the reconstruction of the value after a single failure with just querying the 3 immediate neighbors. This implies a tradeoff between storage space and recovery capability, which could be further investigated.

Area Failures

Area failures introduce more complication than isolated node failures, but they can still be addressed. Known areas of failure translate into infinite weights in the transformation graph described in section 6.2.1. Thus the value of a failing area can be calculated by querying on that area and appropriately setting as infinite the weights in the transformation graph.

Reconstructing the value of an entire failed area is not always necessary, as failing areas can just be bypassed during plan construction for other queries, by again setting the appropriate edges to infinite weights. So, if a query region only intersects a part of the failing area setting the edge weights appropriately will force the planning algorithm to avoid the failed areas.

Note that areas can fail in such a way that the failures cannot be bypassed for some queries. For the query of Figure 6.3, if areas 2 and 4 fail, there is no way of retrieving the exact answer. We can compute the total lost value in the combined $\{2,4\}$ area, but the query only intersects part of that region, and we can't compute the portion of that value that corresponds to just those areas.

in the cases where computing the accurate answer is infeasible, we want to be able to approximate it in the best way possible. Without knowledge of correlations or any other information on the actual data, the best that we can do is to assume a uniform distribution over the values in the failed area, and use it to estimate the aggregate in a portion of it. This estimate will be more accurate, as the failed area becomes smaller relative to the region of interest. Therefore, in order to compute estimates over a failed region of interest, we should retrieve the aggregate of minimal area A, such that $A \supseteq Q_A$. If $A = G_A$ then the query can be computed accurately.

Figure 6.12 shows an example where the query value can be computed accurately. Algo-

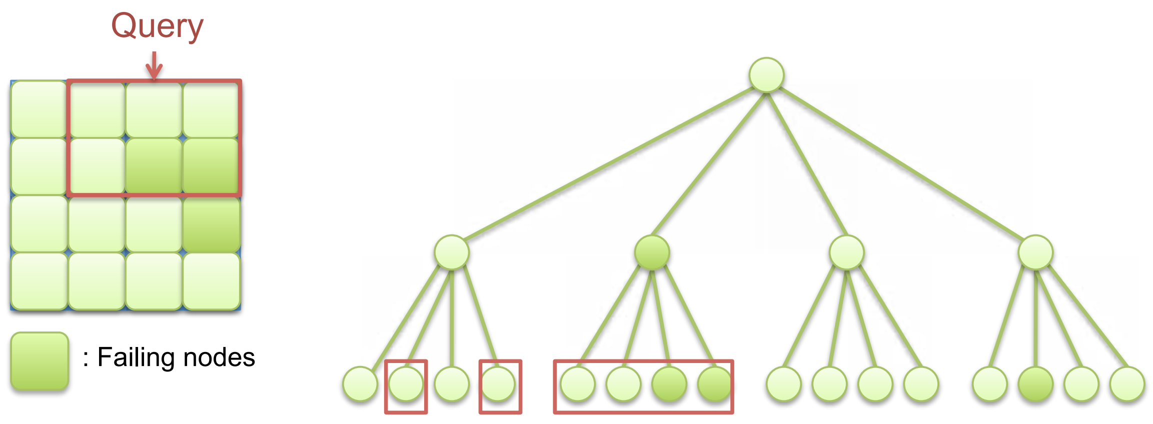


Figure 6.12. Example query that can be computed accurately

rithm 13 computes the aggregate over a failed region, or returns an estimate if that cannot be computed accurately.

Figure 6.13 shows an example where the query value cannot be computed accurately, since as we move up from the leaf level, a new failed node is included as a sibling of the previous failed area. Note however that by using Algorithm 13 computation of the estimate is based on just the two failed sibling nodes, and not all 3, producing a more accurate result.

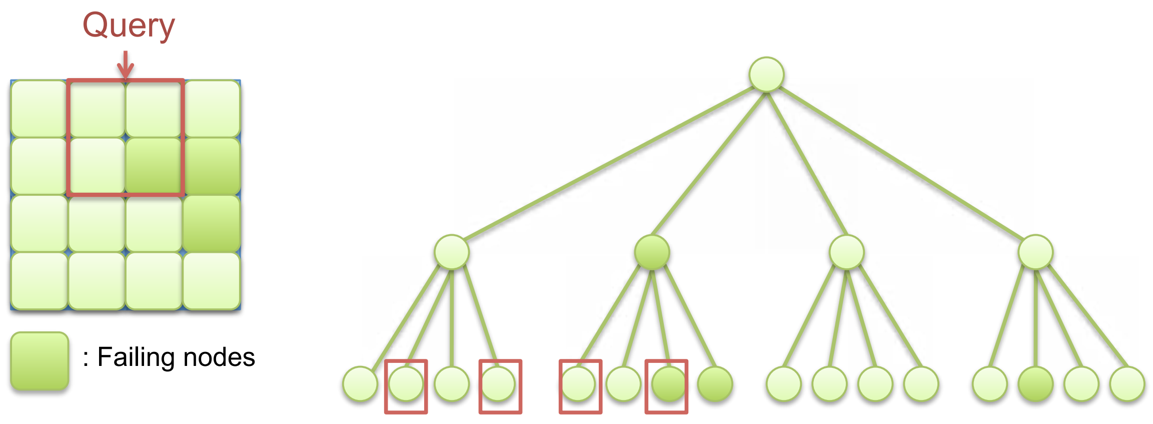


Figure 6.13. Example query that cannot be computed accurately

Algorithm 13 Region Recovery

```
Start at leaf level failed regions $Q_A$. Initialize $V = 0$ and $A = Q_A$;
while fail(currentNode) do
  level−−;
  if $fail(children(currentNode)) > 1$ then
    $A = A \cup newLeafFailures$;
    $V = V - \sum V(aliveLeavesInNewFailure)$;
  end if
  $V = V - \sum V(aliveChildren)$;
  if $!fail(currentNode)$ then
    $V = V + V(currentNode)$;
  end if
end while
if $A > Q_A$ then
  region $Q_A$ cannot be fully recovered
end if
return $estimate = V\frac{Q_A}{A}$
```

6.3 The Grid as an Overlay

In Section 6.2 we discussed the evaluation of spatial aggregate queries using a multiresolution cube hierarchy constructed over a grid network. We developed algorithms for the hierarchy construction and query planning, based on deterministic data at the grid locations. Even though actual grid deployments are not common, a grid network can always be used as an overlay over any deployment (Figure 6.14)

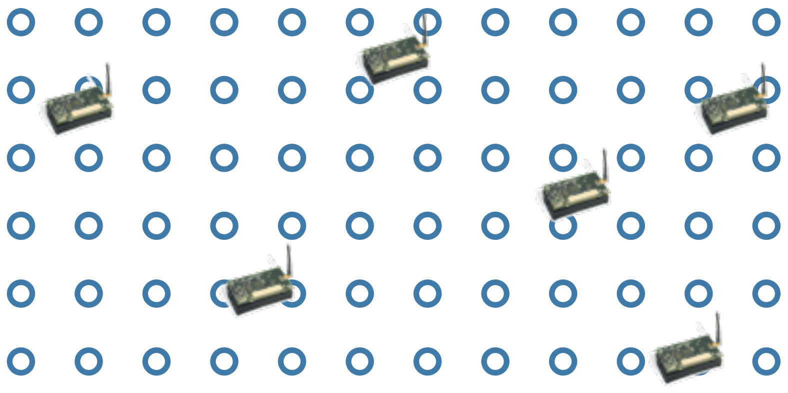


Figure 6.14. The grid doesn't have to be an actual grid deployment, but an overlay over the real deployment

Using observations at the actual sensor locations and spatial models of the data distributions, we can infer the values at the grid locations. The values however will no longer be deterministic, but we will have a mean and a variance value associated with every point. A gaussian distribution with these parameters can naturally represent our belief of the values at the grid locations, similar to the approach followed in Chapter 5.

6.3.1 Summarizing Uncertain Data

Summarization across the grid preserves the gaussian family, as any linear combination of gaussians is also a gaussian. The core of the summarization schemes can be the same as in the deterministic case, however a number of new challenges arise. If we assume independence across the grid points, then the case of simple sums across each cell is straightforward:

$$V(C) = \sum_i V(c_i)$$

$$\sigma^2(C) = \sum_i \sigma^2(c_i)$$

The case of prefix-sum becomes complicated even with the independence assumption. The reason is that even with the individual grid locations being independent, the partial summations along the grid dimensions introduce correlations between the partial sums. Figure 6.15 shows how the prefix-sum value of nodes on the grid can share overlapping regions. The prefix-sum values of those two grid location have a non-zero covariance.

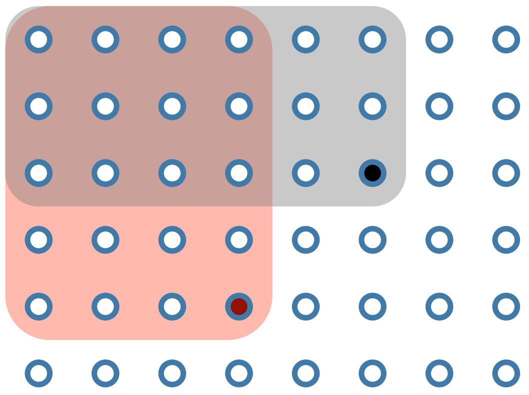


Figure 6.15. In PS, partial sums share grid regions and are therefore correlated.

The mean values at each location can always be updated using the construction algorithm presented in Section 6.2.4. So, for the PS propagation shown in Figure 6.16, the mean value at d would be calculated as $V(D) = V(B) + V(C) - V(A) + V(d)$, where with small letters we denote the local value, and with capital letters the prefix-sum value at the corresponding location. For the variances however, it is no longer true that $\sigma^2(D) = \sigma^2(B)+\sigma^2(C)+\sigma^2(A)+\sigma^2(d)$, because A, B and C are no longer independent (even if a, b and c are).

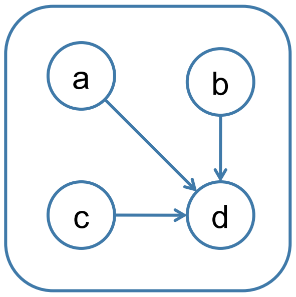


Figure 6.16. In the prefix-sum algorithm data is propagated across 2 dimensions.

The right equation would be

$$\sigma^2(D) = \sigma^2(B) + \sigma^2(C) + \sigma^2(A) + \sigma^2(d) - 2Cov(A,B) - 2Cov(A,C) + 2Cov(B,C)$$

This would require the computation and propagation of covariances for the cube construction. Alternatively, we can just have a scheme that propagates the variances of all locations corresponding to a sum, as ultimately $\sigma^2(D)$ would be equal to $\sum_i \sigma^2(c_i)$, where c_i all the cells in the area covered by the prefix-sum D. Regardless, either approach requires maintaining and sending a lot of data over the network.

Things become even more complicated when we relax the independence assumption, which was unreasonable in the first place if the grid values are inferred from a network overlay. Note that the computation of the appropriate covariance matrices, and the construction of the hierarchy assuming non-independent grid values, is feasible if done centrally by gathering all the data. A distributed approach however becomes challenging, as all the covariance values cannot be stored at all grid locations.

Some first steps for future work to address this problem would be to only account for local dependencies, assuming that far away nodes are not correlated, which is a reasonable assumption in many settings, or only compute covariances across cell borders.

Chapter 7

Conclusions and Open Problems

7.1 Summary

Sensor networks find use in a variety of applications including habitat monitoring, ecological studies, environment observation and forecasting, structural monitoring, and various home and business applications. Increasing the network lifetime is a major concern in most of these settings, as replacing depleted batteries is often not a viable option. In order to address this fundamental issue we aim to optimize the common task of monitoring queries in terms of energy consumption. Monitoring queries invoke the use of sensory devices as well as the wireless medium, both of which are the most energy demanding components of sensor nodes.

In this work we study methods to optimize the use of these components during query propagation and data gathering, using both centralized optimization as well as distributed techniques. This thesis revolves around efficient querying in resource constraint settings. Our approach involves centralized planning, as well as in-network techniques.

In Chapter 3 we tackle the problem of communication cost optimization. When we deal with instances of selective data gathering, like it is the case in the BBQ system [28], traditional flood-based techniques behave poorly. This is because many more nodes are involved in communication than it is actually necessary. Flood-based methods are good when a large portion of the nodes are involved in the query, but in more selective scenarios we need more specialized retrieval mechanisms.

We proposed the interleaving of query propagation and data gathering into a single phase, managing to transform the problem into a TSP variant: the Minimum Splitting Tour Problem. Finding the optimal splitting tour is an NP-hard problem, however we can approximate it within a constant factor. First of all we proved that the optimal simple tour (TSP) approximates the optimal splitting tour with a factor of 1.5: $Cost_{TSP} \leq 1.5 Cost_{ST}$. But solving the TSP is also hard. However, for TSP with triangle inequality, Christofides's algorithm provides a 1.5 factor approximation. Using the same algorithm for the minimum splitting tour problem we proved that we get a 1.75 factor approximation for the optimal splitting tour: $Cost_{T\hat{S}P} \leq 1.75 Cost_{ST}$.

In Chapter 4 we combine the informativeness and communication cost into a joint optimization problem and we addressed the issue of temporal planning in the case of continuous queries. A continuous query is defined by a regular approximate answer query which will be repeated two or more times in predefined intervals. The knowledge of this repetition can provide some additional leverage in the process of optimization. Gathering of information should take into account future repetitions, as extra work in earlier timesteps can ameliorate the cost of future ones.

At the same time, cost and information should not be treated as separate metrics, but they should be combined into a joint optimization problem. For example, a highly informative node which is hard to reach may be less desirable than less informative nodes with easier access.

We transform the multiple timestep problem into an equivalent single-step one, with the introduction of the non-myopic planning graph. Even after this reduction, we still have to solve the single step problem. An important observation that led to our solution is the diminishing returns property of the information function: adding an observation to set A provides more information increase than adding it to a superset B.

$$F(B \cup X) - F(B) \leq F(A \cup X) - F(A) \quad (A \subseteq B)$$

What that means is that the information function is *submodular*. The submodularity property points us to a practically dual problem, the Submodular Orienteering Problem. Our approach uses the SOP solution as a blackbox, and through a covering algorithm transforms it into a solution for our planning problem. The complexity of this approach increases with $O(T^{\log T})$ with the number of timesteps, as the algorithm is executed on the

NPG. We improved on the time complexity by introducing a greedy algorithm which can run on the non-expanded network graph, with overall complexity $O(B^2T(nB)^{\log n})$.

In Chapter 5, we develop an in-network approach to make routing decisions and collect data based on neighborhood summaries. To answer queries in a sensornet setting using information residing in the network, we introduce “in-network summaries”, or “spanning summary trees”. An in-network summary is a spanning tree of the network, in which every node stores a model of the data in each of the subtrees it points to. As the subtrees become smaller in the lower levels of the hierarchy, the models become finer and more precise. A query using that hierarchy can explore the structure starting at the root and going only as deep as is necessary to provide answers of good quality. We optimized this tree structure with the objective of minimizing the communication cost of answering queries. The guarantee achieved is that for n data nodes the query will produce on expectation δn answers that are within w distance of their actual value.

Assuming for simplicity that T has fixed fanout F, data resides at the leaves represented by the single gaussian distributions, and every internal node keeps F pairs of (childptr, gaussian k-mixture), then our “data structure” closely resembles a database index. Given a specified spanning tree T, we can imagine “bulk loading” the models M_v with a bottom-up construction, combining gaussian mixtures as we climb up the hierarchy. In this approach, mixtures high in the tree will be quite large. Since we need to keep the size restricted, we need to have a method for compressing the mixtures, otherwise called “collapsing”.

In the problem of compression, our input is a mixture (set) of l gaussian distributions, and our output a k-size mixture, $k < l$. The parameter k is dictated by the amount of storage space assigned to the model on every node, and is not required to be the same on each one of them. We first address the problem for the case of $k = 1$, assuming that the summary hierarchy keeps a single gaussian distribution as a model of all the sensors in each subtree.

We observe that collapsing distributions poses unique challenges in our setting that cannot be accommodated by traditional approaches, like using a distance metric like KL-divergence. We develop a compression algorithm based on a query workload, and we show how to build summary hierarchies and how to evaluate a “`SELECT *`” query over one.

Finally, in Chapter 6 we study query evaluation and in-network summary structures for spatial aggregate queries. As an extension of in-network modeling and in-network summaries, we introduced multi-resolution data cubes. Multi-resolution cubes store area aggregate information at different levels of granularity, and can be used to answer spatial aggregate queries. Spatial aggregate queries define an area of interest G over a grid of nodes. Using the hierarchical cube summaries we can extract the aggregate information for area G.

We show that the optimal query plan over a multiresolution cube can be easily computed through a transformation to a max-flow min-cut problem. We further extend the algorithm to support optimization over multiple queries. Also the hierarchy can by augmented with a richer summary scheme (e.g. prefix sum cubes) which provide the potential for cheaper plans. One interesting problem in this setting is the study of failures. We show that single node failures can be easily overcome using information from neighboring nodes, and we also presented methods to recover from area failures.

Overall this thesis focuses on the problem of reducing communication during query execution in sensor networks, and does so through centralized and distributed techniques. The ideas can be generalized to other environments with resource restrictions, where sensornets are a representative application.

7.2 Limitations and Future Directions

Throughout the course of this thesis we addressed a number of hard problems, but also unveiled various interesting issues that remain unresolved, some of which could lead to interesting research topics.

7.2.1 The Communication Model

In Chapters 3 and 4 of this thesis, we assumed a static model of the network topology. We modeled it as a graph with appropriate edge weights to represent link quality. Considering that the goal is to minimize communication cost of the constructed query plans, the success of our algorithms heavily relies on the quality and representativity of this model. For topologies that are relatively static and where the link qualities do not change erratically, our schemes perform quite well, and the graph model, albeit simple, provides a rough estimate of how the communication network is expected to perform.

We have to point out however that such a graph representation does not capture the full capabilities of the sensors' communication system. For example, a node's radio can be tuned to transmit at a larger radius at the cost of more energy, expanding the node's reach and improving its link quality with its neighbors. This characteristic is not represented in a network graph with static weights. It would be an interesting problem, not only to find a graph model that captures that, but also study a scheme that automatically adjusts radio power based on link feedback, query routing and failures. The radio tuning technique could possibly also be used for load balancing: nodes that have been getting a lot of routing load can decrease their radio strength, gradually forcing the network to find alternative paths.

Another feature that is missing from the current communication model is the broadcasting ability. In the network model, a node is connected to its neighbors with constant edge weights, representing the average number of messages that we are expected to spend for successful communication over a link. When building routing paths and tours, a message send by a node to multiple directions "pays" the cost of transmission multiple times in the model: we add the cost to send over all the chosen directions. However, due to broadcasting, it is possible that the message could be routed along multiple paths through a single transmission. Note that this does not mean that for the case of splitting tours for example our tour costs are miscalculated. When a tour splits, different packets containing different paths need to be sent in each direction, justifying the multiple transmission. This is however a capability that is not captured by the current model, and could possibly lead to improved solutions.

7.2.2 Failure Handling and Recovery

Due to the communication model not being able to capture sudden changes, and since nodes are often unreliable, failure handling and recovery are crucial parts of any type of path planning. Some recovery algorithms are proposed in Section 3.6, but they are heuristic in nature, and even though they are shown to behave well in practice, they offer no theoretical guarantees. The obvious problem that arises is whether we can improve on failure recovery.

One approach would be to try to build more resilient paths during query optimization. In our schemes, only communication cost and informativeness are used as objectives for path selection. In part, likelihood of failure gets reflected on the communication cost, but that may not always be the case. It is probable that there exist patterns for failures which we could learn and avoid. Assume for instance that temperature or humidity fluctuations throughout the day render nodes at certain locations more likely to fail. We could then exploit that knowledge to avoid those nodes during planning.

Another observation is that a general recovery method may not be the most efficient tactic in all cases. A different approach is to built alternative routes and inject them into the network along with the planned route. Upon failure, the node has a contingency plan ready which is tuned to the specific path, location and type of failure, and which would probably work more efficiently than a general purpose recovery technique.

Also the plan of action could depend on how far along the path the failure occurred. Even without a fine-tuned custom-made recovery plan, there are some general techniques that could make recovery more adaptive. The mere location of the failure (further/closer to the basestation) may make some approaches more rewarding than others. Finally, recovery information could be stored inside the network, and used at a point of failure to get the route back on track.

7.2.3 Data Dependencies

The types of phenomena monitored by sensor network applications usually demonstrate correlations that we can exploit to improve on tasks like query performance, as has been the theme of the first chapters of this thesis. In Chapter 5 however we do not take advantage of those dependencies. The in-network models are built with an assumption of independence. Even so, our algorithms are shown to perform very well. It would be an interesting topic though to develop schemes that take advantage of the data correlations to further improve on performance.

Descent into the lower levels of an in-network summary could use the gathered information to improve on models of other branches, and possibly prune out paths. Moreover, taking correlation into account could improve the compression scheme leading to better models.

Data dependencies have also been discussed in Section 6.3, were there is obviously ground for interesting research. The main issues have already been raised: how do we compute correlations in a distributed fashion; where and how do we store them; which nodes need to know which correlation values; are there parts of the covariance matrix that we can prune out; how should the cube construction algorithm change. The problem is straightforward to solve centrally, but moving the process inside the network is a challenging and interesting problem.

Even more interesting becomes the problem of query answering and failure handling. When dealing with uncertain data, the objectives for constructing a query plan could change. Instead of just minimizing the number of elements that need to be retrieved, we may also want to minimize the uncertainty of the aggregate result. Similarly, in the occurrence of failures, instead of reconstructing the data based on the cube summarization protocol, we can use data dependencies to estimate the lost elements based on another part of the network.

7.2.4 Other Directions

Even though sensornets may seem like a very specific application domain, they are a first step at the exploration of optimization techniques for various resource constrained settings and systems with noisy data. In many applications, resources like bandwidth, computation, energy, storage, come at a cost. Learning and inference techniques can be similarly utilized in such settings to optimize common tasks over the given resource constraints. The problems that we talk about in this section do not originate from this thesis, but are examples of how methods and ideas used here could possibly be applied in different settings.

Modeling and indexing ideas can be adapted into the design of a data compression scheme for other domains (e.g. data centers), where bandwidth can become an issue over large amounts of data. Approaches similar to our compression methods presented in Chapter 5 could be adapted to optimize storage based on query workload. If some types of queries are more common than others then data storage can be altered to make access for those more efficient.

Other applications that combine elements from the presented work are failure detection of various devices, and intrusion detection. Application domains can be of small or large scale, involving data from sensing (humidity/temperature on various system components), and/or logs. Using probabilistic models and inference, that information can be useful to detect or even predict failures of system components, which can provide crucial gains to costs for data and system recovery after failures. Also in this case, as in most monitoring applications, challenges are introduced by the vast amounts of data that are produced, the presence of noisy information, as well as the necessity to reason over data from different sources.

A type of restricted resource that could also be interesting to tackle from the perspective of this thesis is privacy-sensitive information. An example of such a case arises in the scenario of virtual enterprises. Organizations often cooperate, sharing technologies, resources and data. However this data is often sensitive in the sense that a company may not wish to have it freely available to its partners, but *some* sharing is necessary for the partnership. This is a case where data, even though available, is constrained by each company's privacy requirements. Albeit of a different nature, these constrained settings share a lot of common ground with the previous scenarios. Data now becomes a restricted resource, and query results can be optimized based on the given privacy restrictions, and networks and storage can be made more effective given specific workloads and constraints.

7.3 Conclusions

With this thesis we have contributed to the problem of query-centric data retrieval in sensor network settings. We identified communication as a deciding factor of a query's energy usage and developed centralized and distributed techniques to reduce it. In addition to our contribution, we have touched upon open questions that could provide ground for more interesting work. Our algorithms and techniques provide a solution to the problem of query optimization in sensor networks, which is a critical factor for the lifetime of many deployments.

References

[1] *http://alertsystems.org.*

[2] *https://mirage.berkeley.intel-research.net/.*

[3] *Biosensor for toxic detection and process control in nitrification plants*, Journal of Environmental Engineering (2005), 658–663.

[4] D.P. Agrawal, M. Lu, T.C. Keener, M. Dong, and V. Kumar, *Exploiting the use of wireless sensor networks for environmental monitoring*, Environmental Management magazine (2004), 35–41.

[5] Anastassia Ailamaki, Christos Faloutos, Paul S. Fischbeck, Mitchell J. Small, and Jeanne VanBriesen, *An environmental sensor network to determine drinking water quality and security*, SIGMOD Rec. **32** (2003), no. 4, 47–52.

[6] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, *Wireless sensor networks: a survey*, Computer Networks (2002).

[7] Deborah Estrin Lewis Girod Michael Hamilton Alberto Cerpa, Jeremy Elson and Jerry Zhao, *Habitat monitoring: Application driver for wireless communications technology*, 2001 ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean (San Jose, Costa Rica), April 3-5 2001.

[8] Analog Devices, *Data sheet: Small, low power 3-axis ±3g imems accelerometer adxl 330.*

[9] Seung Jun Baek, Gustavo de Veciana, and Xun Su, *Minimizing energy consumption in large-scale sensor networks through distributed data compression and hierarchical aggregation*, IEEE Journal on Selected Areas in Communications (2004).

[10] Xiaole Bai, Santosh Kumar, Ziqiu Yun, Dong Xuan, and Ten H. Lai, *Deploying wireless sensors to achieve both coverage and connectivity*, MobiHoc, 2006.

[11] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger, *The r*-tree: an efficient and robust access method for points and rectangles*, SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data (New York, NY, USA), ACM, 1990, pp. 322–331.

[12] Edoardo S. Biagioni and K. W. Bridges, *The application of remote sensor technology to assist the recovery of rare and endangered species*, International Journal of High Performance Computing Applications **16** (2002), 2002.

[13] Avrim Blum, Shuchi Chawla, David R. Karger, Terran Lane, Adam Meyerson, and Maria Minkoff, *Approximation algorithms for orienteering and discounted-reward tsp*, FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (Washington, DC, USA), IEEE Computer Society, 2003, p. 46.

[14] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri, *Towards sensor database systems*, MDM '01: Proceedings of the Second International Conference on Mobile Data Management (London, UK), Springer-Verlag, 2001, pp. 3–14.

[15] D. Braginsky and D. Estrin, *Rumor routing algorithm for sensor networks*, ICDCS-22, 2002.

[16] David Braginsky and Deborah Estrin, *Rumor routing algorithm for sensor networks*, WSNA, ACM Press, 2002.

[17] W. Caselton and Jim Zidek, *Optimal monitoring network design*, Statistics and Probability Letters (1984).

[18] I-Ming Chao, Bruce L. Golden, and Edward A. Wasil, *A fast and effective heuristic for the orienteering problem*, Eur J Op Res (1996).

[19] Moses Charikar, Samir Khuller, and Balaji Raghavachari, *Algorithms for capacitated vehicle routing*, 30th annual ACM symposium on Theory of computing, ACM Press, 1998.

[20] Chandra Chekuri and Martin Pal, *A recursive greedy algorithm for walks in directed graphs*, FOCS, 2005.

[21] N. Christofides, *Worst case analysis of a new heuristic for the traveling salesman problem*, Tech. report, Carnegie Mellon University, 1976.

[22] R. Cristescu, B. Beferull-Lozano, and M. Vetterli, *On network correlated data gathering*, 2004.

[23] Razvan Cristescu, Baltasar Beferull-Lozano, Martin Vetterli, Deepak Ganesan, and Jugoslava Acimovic, *On the interaction of data representation and routing in sensor networks.*, ICASSP, 2005.

[24] Abhimanyu Das and David Kempe, *Algorithms for subset selection in linear regression*, under review for STOC, 2007.

[25] Douglas De Couto, Daniel Aguayo, Benjamin Chambers, and Robert Morris, *Performance of multihop wireless networks: Shortest path is not enough*, HotNets-I, ACM SIGCOMM, October 2002.

[26] M. Demirbas and Xuming Lu, *Distributed quad-tree for spatial querying in wireless sensor networks*, ICC (2007).

[27] Murat Demirbas and Hakan Ferhatosmanoglu, *Peer-to-peer spatial queries in sensor networks*, P2P '03: Proceedings of the 3rd International Conference on Peer-to-Peer Computing (Washington, DC, USA), IEEE Computer Society, 2003, p. 32.

[28] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong, *Model-driven data acquisition in sensor networks*, VLDB, 2004.

[29] Alexandros G. Dimakis, Vinod Prabhakaran, and Kannan Ramchandran, *Ubiquitous access to distributed data in large-scale sensor networks through decentralized erasure codes*, IPSN (Piscataway, NJ, USA), IEEE Press, 2005, p. 15.

[30] Uriel Feige, *A threshold of ln n for approximating set cover*, J. ACM **45** (1998), no. 4.

[31] Dominique Feillet, Pierre Dejax, and Michel Gendreau, *Traveling salesman problems with profits*, Transportation Science **39** (2005), no. 2.

[32] Deepak Ganesan, Răzvan Cristescu, and Baltasar Beferull-Lozano, *Power-efficient sensor placement and transmission structure for data gathering under distortion constraints*, ACM Trans. Sen. Netw. **2** (2006), no. 2, 155–181.

[33] Deepak Ganesan, Ben Greenstein, Denis Perelyubskiy, Deborah Estrin, and John Heidemann, *Multi-resolution storage and search in sensor networks*, ACM Transactions on Storage (2005).

[34] M. R. Garey and D. S. Johnson, *Computers and intractability – a guide to the theory of np-completeness*, Freeman, San Francisco, 1979.

[35] Naveen Garg, Goran Konjevod, and R. Ravi, *A polylogarithmic approximation algorithm for the group steiner tree problem*, J. Algorithms **37** (2000), no. 1, 66–84.

[36] Minos N. Garofalakis, Joseph M. Hellerstein, and Petros Maniatis, *Proof sketches: Verifiable in-network aggregation.*, ICDE, IEEE, 2007, pp. 996–1005.

[37] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh, *Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals*, J. Data Mining and Knowledge Discovery **1** (1997), no. 1, 29–53.

[38] Carlos Guestrin, Andreas Krause, and Ajit Paul Singh, *Near-optimal sensor placements in gaussian processes*, ICML, 2005.

[39] Ralf Hartmut Güting, *An introduction to spatial database systems*, The VLDB Journal **3** (1994), no. 4, 357–399.

[40] Antonin Guttman, *R-trees: a dynamic index structure for spatial searching*, SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data (New York, NY, USA), ACM, 1984, pp. 47–57.

[41] M. Haimovich and A. H. G. Rinnooy Kan, *Bounds and heuristics for capacitated routing problems*, Mathematics of Operations Research (1985).

[42] Tian He, Sudha Krishnamurthy, John A. Stankovic, Tarek Abdelzaher, Liqian Luo, Radu Stoleru, Ting Yan, Lin Gu and Jonathan Hui, and Bruce Krogh, *An energy-efficient surveillance system using wireless sensor networks*, MobiSys, June 2004.

[43] S. T. Hedetniemi, S. M. Hedetniemi, and A. Liestman, *A survey of gossiping and broadcasting in communication networks*, Networks (1998).

[44] W. Heinzelman, J. Kulik, and H. Balakrishnan, *Adaptive protocols for information dissemination in wireless sensor networks*, 1999.

[45] W. Daniel Hillis and Guy L. Steele, Jr., *Data parallel algorithms*, Commun. ACM **29** (1986), no. 12, 1170–1183.

[46] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin, *Directed diffusion: a scalable and robust communication paradigm for sensor networks*, MobiCom, 2000.

[47] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon, *Health monitoring of civil infrastructures using wireless sensor networks*, IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks (New York, NY, USA), ACM, 2007, pp. 254–263.

[48] Joanna Kulik, Wendi R. Heinzelman, and Hari Balakrishnan, *Negotiation-based protocols for disseminating information in wireless sensor networks*, Wireless Networks (2002).

[49] G. Laporte and S. Martello, *The selective traveling salesman problem*, Discrete Appl. Math. **26** (1990), no. 2-3.

[50] Philip Levis, Nelson Lee, Matt Welsh, and David Culler, *TOSSIM: Accurate and scalable simulation of entire tinyos applications*, Proc. ACM Conference on Embedded Networked Sensor Systems (SenSys), November 2003.

[51] Ming Li, Deepak Ganesan, and Prashant Shenoy, *Presto: feedback-driven data management in sensor networks*, NSDI, 2006.

[52] Xin Li, Young Jin Kim, Ramesh Govindan, and Wei Hong, *Multi-dimensional range queries in sensor networks*, 1st international conference on Embedded networked sensor systems, ACM Press, 2003.

[53] Stephanie Lindsey, Cauligi Raghavendra, and Krishna M. Sivalingam, *Data gathering algorithms in sensor networks using energy metrics*, IEEE Trans. Parallel Distrib. Syst. **13** (2002), no. 9, 924–935.

[54] Juan Liu, Dragan Petrovic, and Feng Zhao, *Multi-step information-directed sensor querying in distributed sensor networks*, ICASSP, 2003.

[55] S. Madden and J. Gehrke, *Query processing in sensor networks*, Pervasive Computing **3** (2004), no. 1.

[56] 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** (2002), no. SI, 131–146.

[57] ———, *The design of an acquisitional query processor for sensor networks*, SIGMOD (New York, NY, USA), ACM, 2003, pp. 491–502.

[58] Samuel Madden, Robert Szewczyk, Michael J. Franklin, and David Culler, *Supporting aggregate queries over ad-hoc wireless sensor networks*, WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications (Washington, DC, USA), IEEE Computer Society, 2002, p. 49.

[59] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong, *Tinydb: an acquisitional query processing system for sensor networks*, ACM Trans. Database Syst. (2005).

[60] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson, *Wireless sensor networks for habitat monitoring*, WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications (New York, NY, USA), ACM, 2002, pp. 88–97.

[61] Gaurav Mathur, Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy, *Ultra-low power data storage for sensor networks*, IPSN, 2006.

[62] K. Mayer, K. Ellis, and K. Taylor, *Cattle health monitoring using wireless sensor networks*, Communication and Computer Networks Conference, 2004.

[63] Alexandra Meliou, David Chu, Carlos Guestrin, Joseph Hellerstein, and Wei Hong, *Data gathering tours in sensor networks*, IPSN, 2006.

[64] Alexandra Meliou, Andreas Krause, Carlos Guestrin, and Joseph M. Hellerstein, *Nonmyopic informative path planning in spatio-temporal models*, AAAI, 2007.

[65] G. Nemhauser, L. Wolsey, and M. Fisher, *An analysis of the approximations for maximizing submodular set functions*, Mathematical Programming **14** (1978), 265–294.

[66] T. Ralphs, L. Kopman, W. Pulleyblank, and L. Trotter, *The capacitated vehicle routing problem.*

[67] C. E. Rasmussen and C. K.I. Williams, *Gaussian processes for machine learning*, Adaptive Computation and Machine Learning, The MIT Press, 2006.

[68] Sylvia Ratnasamy, Brad Karp, Li Yin, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker, *GHT: a geographic hash table for data-centric storage*, WSNA, 2002.

[69] Narayanan Sadagopan and Bhaskar Krishnamachari, *Maximizing data extraction in energy-limited sensor networks.*, INFOCOM, 2004.

[70] Anna Scaglione and Sergio D. Servetto, *On the interdependence of routing and data compression in multi-hop sensor networks*, MobiCom, 2002, pp. 140–147.

[71] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos, *The r+-tree: A dynamic index for multi-dimensional objects*, VLDB, 1987, pp. 507–518.

[72] Ambuj Shatdal and Jeffrey F. Naughton, *Adaptive parallel aggregation algorithms*, SIGMOD Rec. **24** (1995), no. 2, 104–114.

[73] M.C. Shewry and H.P. Wynn, *Maximum entropy sampling*, J Appl Statist **14** (1987).

[74] R. Sim and N. Roy, *Global a-optimal robot exploration in SLAM*, ICRA, 2005.

[75] Amarjeet Singh, Andreas Krause, Carlos Guestrin, William Kaiser, and Maxim Batalin, *Efficient planning of informative paths for multiple robots*, IJCAI, 2007.

[76] Amir Soheili, Vana Kalogeraki, and Dimitrios Gunopulos, *Spatial queries in sensor networks*, GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems (New York, NY, USA), ACM, 2005, pp. 61–70.

[77] Mani Srivastava, Richard Muntz, and Miodrag Potkonjak, *Smart kindergarten: sensor-based wireless networks for smart developmental problem-solving environments*, MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking (New York, NY, USA), ACM, 2001, pp. 132–138.

[78] C. Stachniss, G. Grisetti, and W. Burgard, *Information gain-based exploration using Rao-Blackwellized particle filters*, RSS, 2005.

[79] David C. Steere, Antonio Baptista, Dylan McNamee, Calton Pu, and Jonathan Walpole, *Research challenges in environmental observation and forecasting systems*, MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking (New York, NY, USA), ACM, 2000, pp. 292–299.

[80] Hanbiao Wang, Deborah Estrin, and Lewis Girod, *Preprocessing in a tiered sensor network for habitat monitoring*, EURASIP JASP special issue of sensor networks, 2002, pp. 392–401.

[81] Geoffrey Werner-Allen, Konrad Lorincz, Matt Welsh, Omar Marcillo, Jeff Johnson, Mario Ruiz, and Jonathan Lees, *Deploying a wireless sensor network on an active volcano*, IEEE Internet Computing **10** (2006), no. 2, 18–25.

[82] Alec Woo, Terence Tong, and David Culler, *Taming the underlying challenges of reliable multihop routing in sensor networks*, SenSys, 2003, pp. 14–27.

[83] Weipeng P. Yan and Per-Larson, *Eager aggregation and lazy aggregation*, VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 1995, pp. 345–357.

[84] Yong Yao and Johannes Gehrke, *Query processing for sensor networks*, Conference on Innovative Data Systems Research (CIDR), 2003.

[85] Yang Yu, Bhaskar Krishnamachari, and Viktor K. Prasanna, *Energy-latency tradeoffs for data gathering in wireless sensor networks*, INFOCOM, 2004.