

**Data Gathering Tours in Sensor Networks**

by

Alexandra Meliou

B.S. (National Technical University of Athens) 2003

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

Master of Science

in

Computer Science - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Joseph M. Hellerstein, Chair  
Christos H. Papadimitriou

Fall 2005

---

## **Data Gathering Tours in Sensor Networks**

by Alexandra Meliou

---

### **Research Project**

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

#### **Committee:**

---

Joseph M. Hellerstein

Research Advisor

---

(Date)

\* \* \* \* \*

---

Christos H. Papadimitriou

Second Reader

---

(Date)

Data Gathering Tours in Sensor Networks

Copyright © 2005

by

Alexandra Meliou

## Abstract

Data Gathering Tours in Sensor Networks

by

Alexandra Meliou

Master of Science in Computer Science - Electrical Engineering and Computer Sciences

University of California, Berkeley

Joseph M. Hellerstein, Chair

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 paper we study the algorithmic challenges in efficiently routing a fixed-size packet through a small number of nodes in a sensor network, picking up data as the query is routed. We show that computing the optimal routing scheme to visit a specific set of nodes is NP-complete, but we develop approximation algorithms that produce plans with costs within a constant factor of the optimum. We then enhance the robustness of our initial approach 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.

---

Joseph M. Hellerstein  
Thesis Committee Chair

# Contents

<b>Contents</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Related Work . . . . .	3
<b>2 Optimization</b>	<b>5</b>
2.1 The Optimization Problem . . . . .	5
2.1.1 Optimal communication path . . . . .	7
2.1.2 Problem Definition . . . . .	15
2.1.3 Hardness . . . . .	16
<b>3 Approximations</b>	<b>20</b>
3.1 Approximations . . . . .	20
3.1.1 Bounding the Minimum Splitting Tour with the TSP . . . . .	20
3.1.2 A polynomial approximation for the minimum splitting tour . . . . .	26
<b>4 Heuristics</b>	<b>28</b>
4.1 Packet size limitations . . . . .	28
4.1.1 Path injection . . . . .	28
4.1.2 Cutting a tour . . . . .	30
4.1.3 Multiple packets . . . . .	32
4.1.4 Hybrid: cutting with multiple packets . . . . .	33
<b>5 Recovery</b>	<b>34</b>
5.1 Recovering from failures . . . . .	34
5.1.1 Backtracking . . . . .	35

5.1.2 Flooding . . . . .	36
<b>6 Experiments</b>	<b>39</b>
6.1 Experimental Results . . . . .	39
6.1.1 Simulation Results . . . . .	40
6.1.2 Discussion . . . . .	44
<b>7 Conclusion</b>	<b>46</b>
<b>Bibliography</b>	<b>47</b>
References . . . . .	47



# Chapter 1

## Introduction

In this paper, we consider a basic task in sensor networks: gathering data from a subset of nodes in the network. This problem arises in interactive scenarios, 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. The choice may also be made in software: recent work proposes model-driven querying schemes for sensornets [10], in which an optimization process chooses the set of nodes and sensors to sample in order to approximately answer a high-level SQL query.

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 [23]. 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 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 sensor research is that network connectivity is highly unpredictable. As a result, some protocols assume that individual nodes will maintain routing tables with

their neighbors on an ongoing basis. But 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 [1], as we demonstrate below. In these cases the graph representing the network can be considered *semi-static*. Although the link quality of an edge demonstrates variations over time, its distribution is practically stationary ([27]). This means that its statistical properties do not change much over time (thus our use of the term semi-static) In such situations 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.

In this paper we study the algorithmic challenges lurking behind the apparently simple 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 [19].

## 1.1 Related Work

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 [15] waste bandwidth and energy by making unnecessary transmissions. In a sensor network platform energy restrictions are often very limiting, and the process of data gathering should take energy efficiency into account ([2], [21], [25]). The tradeoff between energy and latency has also been a topic of study ([28]). 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 [7], [8], [26].

The SPIN protocol proposed in [16] and [18] 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 [17] 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 [3], [4] 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, [20] presents the DIM data structure for embedding indices within the sensor network, to allow more efficient retrieval of events. [22] 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 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.

Our work addresses a problem in the BBQ query system [10]. 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 paper 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 [10]. Many applications that rely on selective data gathering could benefit from the theory presented in this paper (e.g., multi-resolution storage [11]). We will make the assumption that the basestation possesses information about the entire network topology, which is assumed semi-static. We do not require the sensor nodes to maintain any routing information, not even for their immediate neighbors.

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 [5], [13], [24]. 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 2.1). 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 explore the latter issue in some detail in our theoretical analysis of Section 2.1.1, though the former distinction is more germane to our actual algorithms.

## Chapter 2

# Optimization

### 2.1 The Optimization Problem

In our setting we have a semi-static sensor network, 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. Every two nodes that can communicate directly without the assistance of others are connected with an edge.

There is also a cost function  $c(u, v)$  representing the expected cost of using the edge  $(u, v)$  in the communication path. This is because not all communication edges are equally good. For example sending a message to a node that is far away has higher probability of packet losses, hence the number of messages we expect to send for successful transmission may be higher. 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 graph is undirected in

our model i.e.,  $c(u, v) = c(v, u)$ <sup>1</sup>. The choice of an undirected model was meant to capture 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 taken in [27] and proposed in [9]. This approach results in an undirected cost graph (the cost is the same in both directions of an edge), but it does not imply symmetry on the link layer.

We assume that periodic link-level measurements are propagated to the basestation to maintain the accuracy of  $G$  and  $c(u, v)$ . 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 5.1.

The measurement set  $R \subseteq V$  can be chosen manually or via software techniques such as those proposed in [10]. Note that although we require measurements from just the nodes in  $R$ , the optimal communication route can contain other nodes that do not belong in the set. The function that we try to minimize is the total communication cost. So, if a link between two nodes of  $R$  is too weak (hence expensive), we may prefer to route the message through a node that may not belong to  $R$ .

Given a network graph  $G$  and measurement set  $R$ , the goal of our optimization problem is to compute a minimal-cost routing scheme that visits all the nodes in  $R$  and brings their data back to the basestation. This 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 4.1 we elaborate 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 for our purposes.

---

<sup>1</sup>Asymmetric links are not unusual in the radios of current sensor networks, but they can be discarded at the networking layer to avoid unnecessary complexity in routing [14].

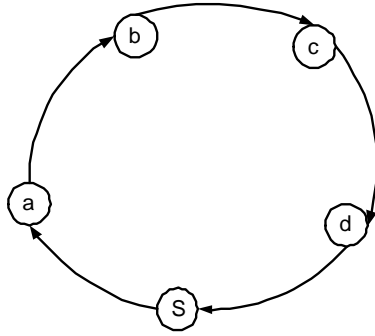


Figure 2.1. Message passing

### 2.1.1 Optimal communication path

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.

For example in Figure 2.1 the base station (node  $S$ ) sends a message to  $a$  containing the whole routing path ( $a \rightarrow b \rightarrow c \rightarrow d$ ). Node  $a$  takes the appropriate measurement and writes its answer in the same message, which it forwards to node  $b$  and so on.

So in this case we get the answers with 5 transmissions, while if we had sent the query first and then wait for the answers, we would need at least 8 transmissions. From now on, we will assume that we follow this message scheme, for which we will give more details later on.

#### Observations

So, in our problem setting, we have a graph  $G(V, E)$  and set  $R$  of nodes of interest. In this section we will examine some simple ideas for querying nodes in  $R$  in an efficient way, and observe where these ideas fail.

Probably the most simple approach is to individually route messages to every node in  $R$ , along the shortest path from the root. But this is not the most efficient thing that one might do. See for example figure 2.2. If we want to query two nodes that are very close together, it is probably better

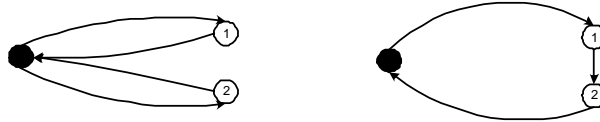


Figure 2.2. Separately route to every node vs making a tour.

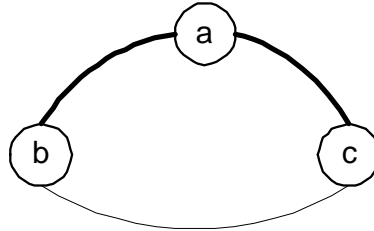


Figure 2.3. MST vs making a tour.

to ask both of them at the same time. The way this is done in this example is that node 1 receives the message processes it and concatenates its answer before forwarding it to node 2.

If we have the possibility of concatenating a node's answer to the message and then forwarding it, then we can optimize globally instead of individually. So, we need to find a subgraph of minimum weight that transfers the message to all nodes and then brings it back to the basestation. If the communication was one-way, i.e. if we just needed to send the message and didn't expect answers, then the obvious solution would be to route over the minimum spanning tree<sup>2</sup>. However, in this two-way communication scheme, the MST approach is not optimal. For example in the simple case presented in figure 2.3, assume that  $c(a, b) = c(a, c) = 1$  and  $c(b, c) = 1.1$ , and that  $a$  is the source. The minimum spanning tree consists of the bold edges. Sending the message on the spanning tree and back would cost 4, while the tour  $a \rightarrow b \rightarrow c \rightarrow a$  costs 3.1.

From the above observations, we can see that looking for an optimal tour can be a reasonable approach to the problem. This is essentially the TSP, with a slight modification. In our case we don't mind going through the same vertex more than once, if of course this gives us a lower total cost<sup>3</sup>(metric TSP).

However we can observe that this approach is not optimal either. In figure 2.4 we have an example graph. We assume that edges that are not presented are of infinite cost. How would we do

<sup>2</sup>In fact when we want to span only a subset of the nodes, what we want is not the MST but the Steiner tree.

<sup>3</sup>From now on, when we refer to the TSP we will refer to this modified version.

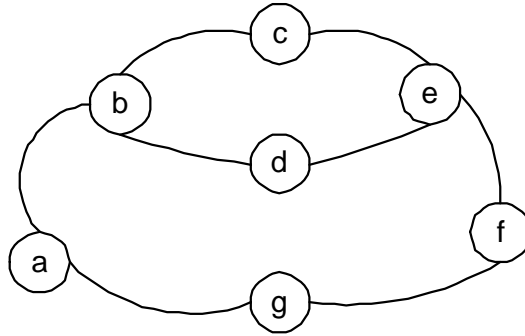


Figure 2.4. TSP vs a more complex tour.

a tour in this case? A solution would be  $a \rightarrow b \rightarrow d \rightarrow e \rightarrow c \rightarrow e \rightarrow f \rightarrow g \rightarrow a$ . But this goes over edge  $(c, e)$  twice. Can we do better than that?

### Splitting Tours

The communication path that we will compute can be 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. We need to figure out what the properties of  $G_s$  should be, in order to satisfy our requirements.

Observing figure 2.4 again we can argue that the problem with the TSP approach is that in some graph cases it has to cross the same edges several times. How can we avoid that? A tour is a simple cycle, it only goes in one direction. The difference with for example the MST case, is that in a tree a path is allowed to branch into two or more separate paths. The optimal routing scheme will be a tour that has the branching property of a tree.

We refer to such a graph  $G_s$  as 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 along the way (e.g., Figure 2.5).

See figure 2.6 for some problems that can come up. Graph (a) cannot be a splitting tour, because a query can never reach node  $a$ , and  $c$ 's answer cannot reach any other node. This example shows that there can be no source or sink components in a splitting tour. For  $G_s$  to be a valid solution to

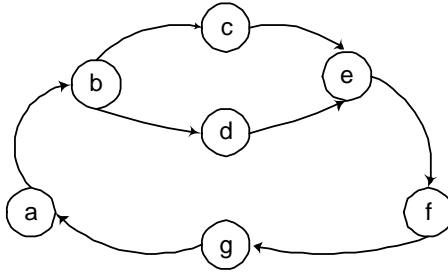


Figure 2.5. A splitting tour, assuming node  $a$  as the basestation. The tour splits at node  $b$  and follows two separate paths which merge at node  $e$ .

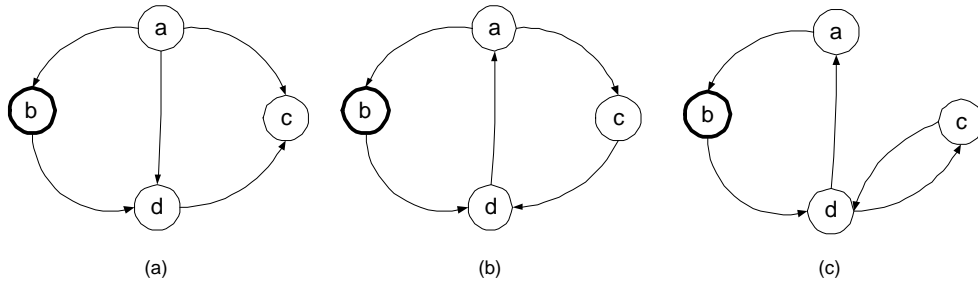


Figure 2.6. Examples of problematic splitting tours (the bold node indicates the basestation).

our problem, it needs to be a strongly connected graph. This means that there should be a path from every node to every other node.

Strong connectivity is a necessary but not sufficient condition. Graph (b) shows a more complicated problem. Although the graph is strongly connected,  $c$ 's data cannot reach the basestation.  $b$  sends the query to  $d$ , which forwards it along with its answer to  $a$ . Then  $a$  can send everything it has back to the basestation, but  $c$ 's answer is missing. If  $a$  forwards the query to  $c$  and then  $c$  to  $d$ ,  $d$  cannot do anything now because it has already used the edge  $(d, a)$ <sup>4</sup>. The problem that appeared in this case is that  $c$  needs to "wait" to be queried through  $b$ , and  $b$  needs to "wait"  $c$ 's answer before using its last outgoing edge. The result is a *deadlock* in the graph.

Now let's look at the case presented by graph (c). The problem appears when  $d$  receives the message from  $b$  and has to decide where to forward it.  $d$  has 2 outgoing edges  $(d, a)$  and  $(d, c)$ . It can either choose to forward on one or both of them. But if it chooses to send the message along  $(d, a)$  before receiving a message on  $(c, b)$  then we get to an equivalent situation with case (b). The

<sup>4</sup>We do not allow re-usage of edges, because we want the graph to be representative of the total cost of the routing.

only choice for  $d$  that gives a valid splitting tour is to forward on  $(d, c)$  when it receives a message from  $(b, d)$ , and to  $(d, a)$  when it receives a message from  $(c, d)$ .

This example indicates that a directed graph can provide a splitting tour, but there is also some additional information which is necessary to guarantee that all nodes will be queried and their data will reach the basestation. To model this concept, we introduce the notion of *routing rules*.

**Definition 1 (Routing Rule)** *A routing rule for node  $u$  is a rule of the form*

$$E_{receive} \rightarrow E_{send}$$

where  $E_{receive}$  is a set of edges coming into  $u$ , and  $E_{send}$  is a set of edges leaving  $u$ .

*The above routing rule means that  $u$  will forward a message over all edges of  $E_{send}$  after receiving messages from all the edges in  $E_{receive}$ .*

In a splitting tour, every node follows a forwarding schedule defined by a set of routing rules

Example:

Node  $u$  has edges  $a, b$  and  $c$  coming into it and edges  $d, e$  and  $f$  leaving it.

$$a \rightarrow d, e$$

$$b, c \rightarrow f$$

is a set of routing rules for node  $u$ . These rules indicate that when  $u$  receives a message from edge  $a$ , it forwards it to edges  $d$  and  $e$ , and when it receives a message from  $b$  (or  $c$ ) it also has to wait for  $c$  (or  $b$ ) before forwarding the message to  $f$ .

**Definition 2** *A node  $u$  waits for a node  $v$ , if there exists a routing rule for  $u$  for which  $v \in E_{receive}$ , and we write that as  $v \hookrightarrow u$ .*

*Waiting is transitive, i.e. if  $v \hookrightarrow u$  and  $u \hookrightarrow w$  then also  $v \hookrightarrow w$ .*

We need to point out that if  $v \hookrightarrow u$  and  $u \hookrightarrow v$ , we don't necessarily have a deadlock. For example, in figure 2.6.(c), it is  $d \hookrightarrow c$  and  $c \hookrightarrow d$ , but we don't have a deadlock.

So, in order to ensure that all nodes in the communication path are able to both receive the query and deliver the results, we need to define the proper routing rules for all nodes in the graph. However, before defining the routing rules, a splitting tour needs to satisfy certain graph properties. A necessary condition is that every cut in the graph is of minimum size  $2^5$ . 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 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 3 (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 4 (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.*

We can now give a more formal definition of a splitting tour as follows:

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

1. *It is 2-edge connected.*

---

<sup>5</sup>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  $v \in V_A$ .

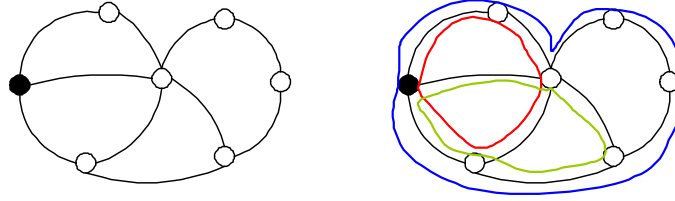


Figure 2.7. Covering a 2-edge connected graph with cycles.

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 the a splitting tour is a 2-edge connected graph is an obvious observation that we have already mentioned. We now need 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. In other words, we need to provide a proper set of routing rules for every node.

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 2.7.

Let's say that  $S = \{C_1, C_2, \dots, C_n\}$  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. Let's 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  $b$  ( $p = a \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_t \rightarrow b$ ). 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  $E_{receive} \rightarrow E_{send}$  of node  $a \equiv b$  and replace it with the rules  $E_{receive} \rightarrow (a, v_1)$  and  $(v_t, b) \rightarrow E_{send}$ . We add the nodes of the cycle to  $V_s$ , and obviously it is still a splitting tour.

Let's look at the case where  $a \neq b$ .

Since we direct every cycle we know that there is at least one edge coming in and one edge coming out of both nodes  $a$  and  $b$ , because  $a$  and  $b$  belong to paths that were previously directed, so  $a, b \in V_s$ . 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  $E_{receive}^a \rightarrow E_{send}^a, (a, v_1)$  for  $a$ , and  $E_{receive}^b, (v_t, b) \rightarrow E_{send}^b$  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.

If  $b \hookrightarrow a$ , then the procedure is inverted, while if none waits for the other, we can direct arbitrarily. The new routing rules inserted in this last case, will now pose an ordering between  $a$  and  $b$  which didn't exist before. 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. QED. ■

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 the minimum cost.

### 2.1.2 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 solution to the minimum Splitting Tour Problem <sup>6</sup>, gives us the best message routing scheme in terms of cost minimization.

The splitting tour problem can be reduced to (is the same as) 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  $R \cap S \neq \emptyset$ , then 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  $S$ .

In the above problem statement, notice that it cannot be  $S = V'$ . obviously in that case, the edges coming in and out should be 0. Although in our case the graph is undirected, we cannot relax the requirement of an edge in and an edge out, to just an edge in. Although we may enter  $S$  on an edge and use the same edge to exit, still we have to count its weight twice <sup>7</sup>.

A splitting tour is a 2-edge connected graph, and any 2-edge connected graph represents a splitting tour.

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

$$\begin{aligned} & \text{minimize } \sum_{e \in E} w_e x_e \\ & \text{s.t.} \end{aligned}$$

---

<sup>6</sup>We may refer to the minimum splitting tour problem as STP.

<sup>7</sup>If that was possible then the problem would reduce to the MST, which would make it polynomial solvable.

$$\forall S : \sum_{\substack{e=(u,v) \in E \\ u \in S, v \notin S}} x_e \geq 1$$

$$\forall S : \sum_{\substack{e=(u,v) \in E \\ u \notin S, v \in S}} x_e \geq 1$$

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  $x_e$  to be 0 or 1, but instead we will get a number between 0 and 1.

### 2.1.3 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 [12]. 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 of 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 we have done 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  $G(V, E)$  as input and an integer  $B$ . 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<sup>8</sup>

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

---

<sup>8</sup>A bridge is an edge whose removal disconnects the graph.

*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.

Let's see 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:** First of all, the minimum cut will be  $\geq 2$  because of the 2-edge connectivity.

Let's 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  $> 2$ <sup>9</sup>.

Since  $G'$  is 2-edge connected, the minimum cut is  $\geq 2$ . 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  $\geq 2$  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.

But 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.

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

Let's say 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

---

<sup>9</sup>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.

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 dfs 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 reduction presented above 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.

Actually 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 is whether the given set spans all the nodes of interest, whether the result is 2-edge connected (no bridges) and the sum of the cost of the edges is  $\leq B$ , which can all 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.

So, since the problem is in  $NP$  and it is NP-hard, it is also  $NP$ -complete. ■

# Chapter 3

## Approximations

### 3.1 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. It is natural as a first step towards this goal, to examine the connection this problem has with a very similar graph problem, which 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.

#### 3.1.1 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 <sup>1</sup>.

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.,  $C_{MSTP}^{opt} \leq C_{TSP}^{opt}$ . 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.

**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).*

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

**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.

---

<sup>1</sup>Note that we do not mind going through the same node more than once

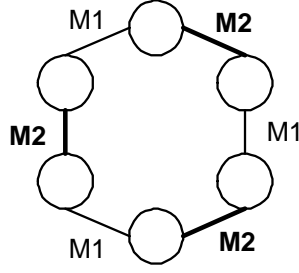


Figure 3.1. A tour  $T$  through an even number of nodes defines two matchings between these nodes,  $M1$  (non-bold edges) and  $M2$  (bold edges).

**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<sup>2</sup> between the nodes  $V_{odd}$ . The length of the tour is the sum of the length of the matches (see example in Figure 3.1).

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

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:

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

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

<sup>2</sup>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.

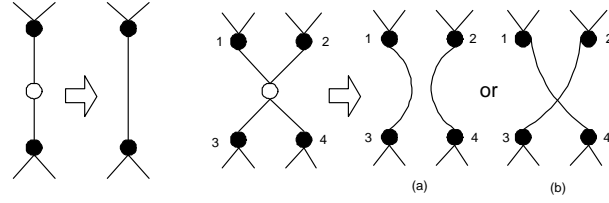


Figure 3.2. Shortcutting even degree nodes.

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.

For 2-degree nodes there is only one way to shortcut them, as shown on the left of Figure 3.2. When the node has higher degree, we can shortcut arbitrarily (see again Figure 3.2), as long as the graph remains connected.

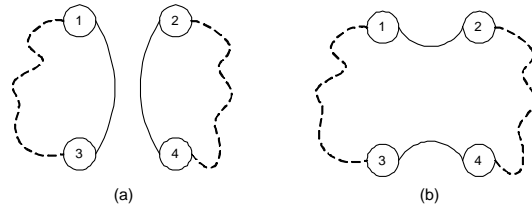


Figure 3.3. Shortcutting 4-degree nodes.

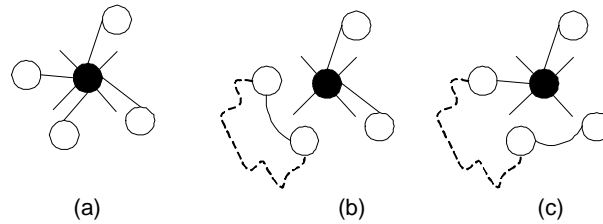


Figure 3.4. Shortcutting  $k$ -degree nodes.

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

Suppose for example that the shortcutting scheme (a) of Figure 3.3 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.

Let us now see what happens in the case where we want to shortcut a node with an arbitrary even degree.

Suppose we want to shortcut the black node of Figure 3.4 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

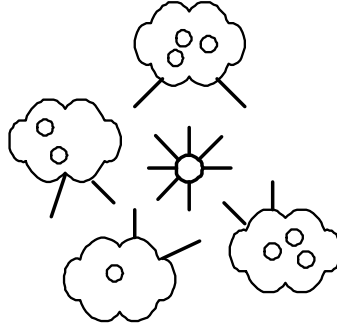


Figure 3.5. If each subset has exactly 2 edges coming out of it, then the total number of edges due to subsets is even, while the edges coming out of the odd degree node is an odd number. Totally we get an odd total number of "incomplete" edges, which cannot be paired.

nodes for which only one edge went in/out of the set. If that was so, 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, in 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.

So, how do we know that for every odd degree node there will always exist a proper edge  $(u, v)$  to choose in the above procedure?

Let's say 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 3.5

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<sup>3</sup>. Therefore, 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.

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  $v \in V_o$ . ■

### 3.1.2 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<sup>4</sup>, which runs in  $O(n^3)$  time [6] 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 [6] 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  $C_{T_2} = C_{T_1} + C_M \leq C_{TSP} + \frac{1}{2}C_{TSP} = 1.5C_{TSP}$
- 

<sup>3</sup>Handshaking Lemma: *In any graph, the sum of all the vertex-degree is equal to twice the number of edges.*

<sup>4</sup>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.

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.

**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, so it is straightforward that:

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

To see why the above is true, name the graph defining the splitting tour  $G_s$ , which is a subgraph of the original graph  $G$ , spanning all the nodes of interest. The minimum spanning tree of graph  $G_s$  is  $T'_1$  and it is  $C_{T_1} \leq C_{T'_1} \leq C_{MSTP}^{opt}$ .

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

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

Therefore

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

■

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.

# Chapter 4

## Heuristics

### 4.1 Packet size limitations

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 4.1.1, and then three schemes for dealing with long tours in Sections 4.1.2 through 4.1.4.

#### 4.1.1 Path injection

In Section 2.1 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 4.1. The packet header 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

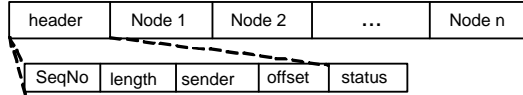


Figure 4.1. Packet structure.

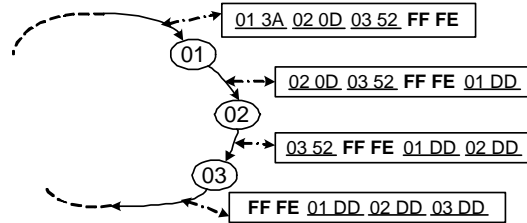


Figure 4.2. Example of how the packet changes from hop to hop. Two bytes are allocated per node. The first one represents the nodeID and the second holds the necessary data to instruct the node whether it needs to sample or not, how many retries it should attempt for the next hop etc. A byte with the 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 delimiters that separate the routing information from the data storage.

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 delimiters can be used. An example of how the packet gets routed is given in Figure 4.2.

The advantage of using this dynamic scheme compared to a static one-slot allocation per 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 4.1.3.

### 4.1.2 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.

Let us say 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. Cuts in the tour are going to be performed by re-routing intermediate edges to the source.

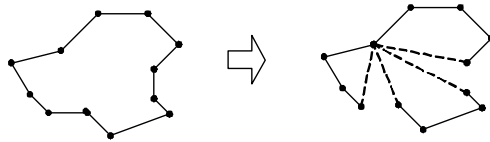


Figure 4.3. Cutting a tour into smaller subtours.

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 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  $i$ th 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.

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  $j$ :

$$S \rightarrow \dots \rightarrow i \rightarrow \dots \rightarrow j \rightarrow \dots \rightarrow S.$$

---

**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
```

---

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  $T_G = S \rightarrow a \rightarrow B \rightarrow c \rightarrow d \rightarrow E \rightarrow f \rightarrow S$ , 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.,  $S \rightarrow a \rightarrow B \rightarrow g \rightarrow E \rightarrow S$ . 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  $S \rightarrow a \rightarrow B \rightarrow c \rightarrow S$  and  $S \rightarrow d \rightarrow E \rightarrow f \rightarrow S$ , 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.

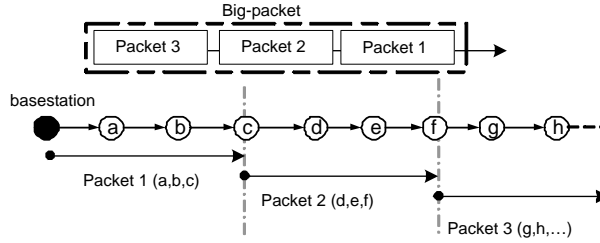


Figure 4.4. Every individual packet holds information for some part of the route. All of them combined can behave like one big packet that holds the whole path and traverses it. Note that during hops data gets transferred between one packet to another, because they all together form a big cyclic buffer.

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  $O(n^3P)$ .

### 4.1.3 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 4.1.1, 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.

#### 4.1.4 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 4.1.2, setting the packet size to  $P$ ; then, we run the same algorithm with a packet of size  $2P$ , computing the edge costs accordingly<sup>1</sup>, then for  $3P$ , and so on. Finally, we define  $L(i, j)$  to be the minimum over all of these packet size options.<sup>2</sup> 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 6.1 we will assess the merits of the various schemes in practice on a real network graph.

---

<sup>1</sup>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.

<sup>2</sup>We can also use a modified version of the DP algorithm to compute the multiple packet version of  $L(i, j)$  more efficiently.

## Chapter 5

# Recovery

### 5.1 Recovering from 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.

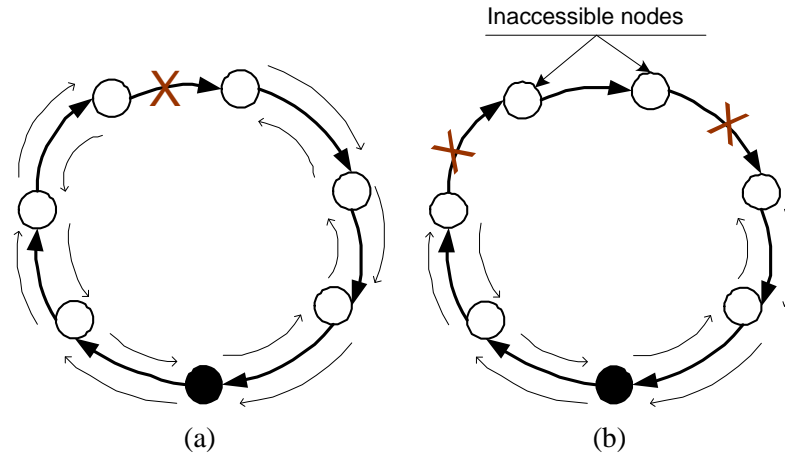


Figure 5.1. The bold edges indicate the initially computed tour. (a) During the traversal a failure is encountered and the message backtracks to the root; a new message is issued in the opposite direction than the tour was defined to gather data from the unvisited part. (b) In case of multiple failures nodes can become inaccessible.

### 5.1.1 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 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 5.1(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 5.1(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.

### 5.1.2 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 5.2.

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

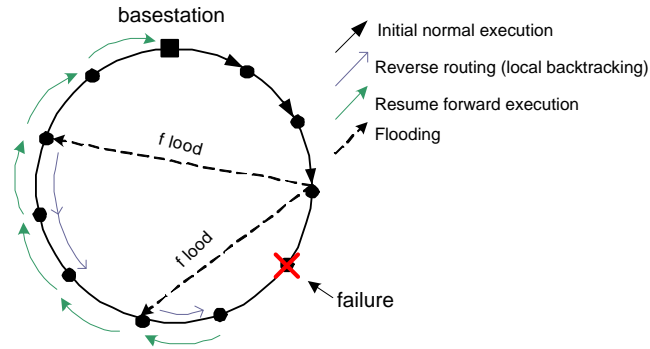


Figure 5.2. When a node detects a failure on the path it initiates a flood with small depth, so that it will remain local. The nodes in the unvisited part of the path that hear the flood backtrack on the path to get any data possible between the failure and their position. If a forward and a backtracking message meet, the backtracking one is killed.

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.

## Chapter 6

# Experiments

### 6.1 Experimental Results

We evaluated our proposed schemes via an implementation, which consists of two separate components. The first involves several Matlab routines used to perform the optimization described in Section 3.1, as well as to apply the packet size restriction of the network, using the algorithms presented in Section 4.1. 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.

The connectivity data given as input to the Matlab code was gathered by running TinyDB queries in a live network for a number of epochs. We chose to use the public mote testbed at Intel Research Berkeley, which is remotely available via the Mirage resource allocation system [1]. At the time, the testbed consisted of 96 Mica2 nodes at fixed positions<sup>1</sup>. 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). We analyzed the connectivity data from sev-

---

<sup>1</sup>These were recently replaced by MicaZ motes.

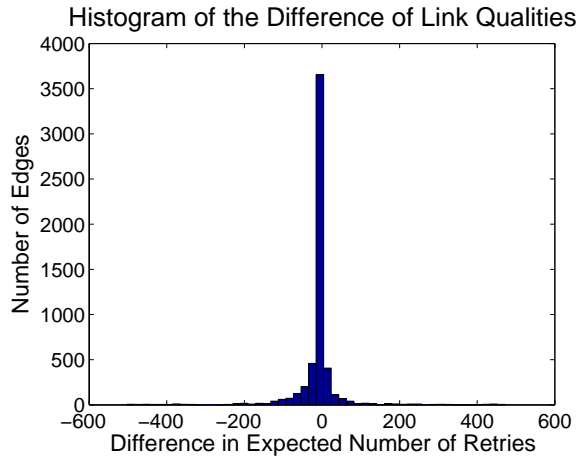


Figure 6.1. Histogram of the difference of the expected number of retries on each edge, for 2 runs of TinyDB placed one day apart.

eral different runs of TinyDB that were executed one or more days apart, and concluded that the semi-static topology is a reasonable assumption (see Figure 6.1).

### 6.1.1 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 [19]. 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 3.1. The main goal of our analysis is to compare the heuristics that we proposed in Section 4.1, 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 6.2 demonstrates how the communication cost of

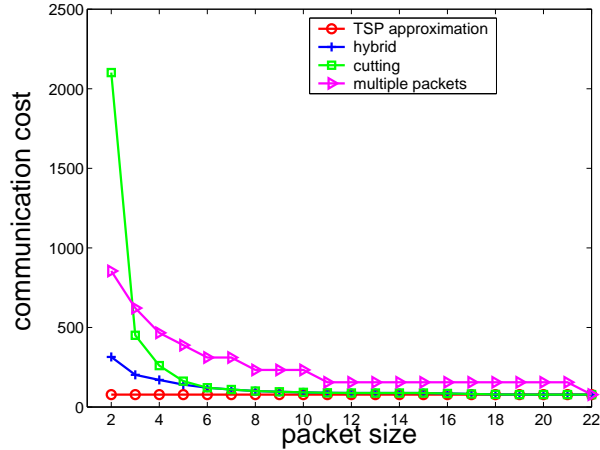


Figure 6.2. Communication cost of the 3 packet adjustment algorithms. This particular graph corresponds to a measuring set of size 15 in a network of 54 nodes.

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 6.2 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 6.3 and 6.4. 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.

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 6.5 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. These TOSSIM runs were performed by using packets of size 30 bytes. In

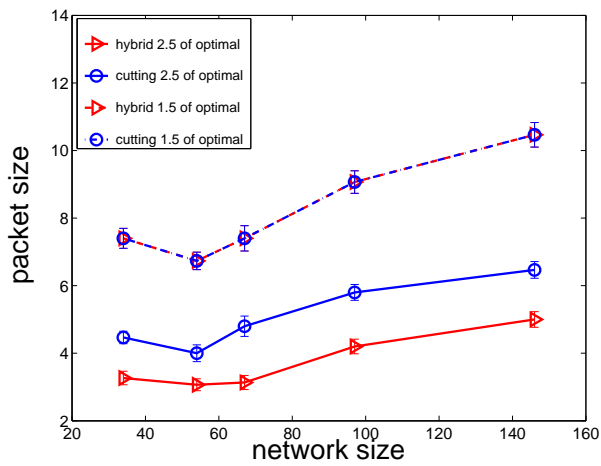


Figure 6.3. Packet size required for reaching a constant factor of the optimal cost, for networks of different size.

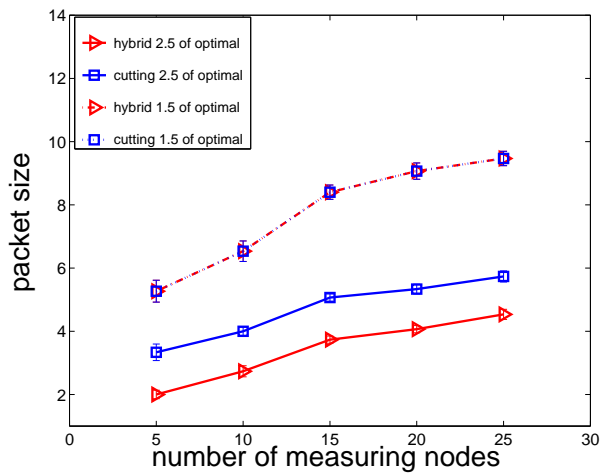


Figure 6.4. Packet size required for reaching a constant factor of the optimal cost for measuring sets of different size.

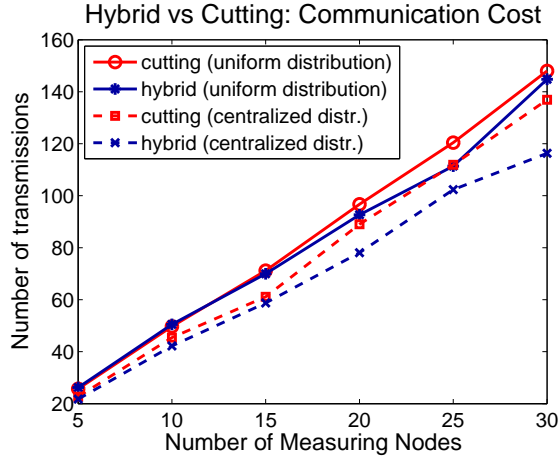


Figure 6.5. Comparison of the cost of the cutting and hybrid heuristics for measuring sets of various sizes chosen by two different distributions from all the network nodes.

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 <sup>2</sup> in TOSSIM a bigger flooding depth didn't seem to add value to the recovery and even started to cause interference phenomena.

Figures 6.6 and 6.7 present experimental results for the two recovery approaches, corresponding to a 5%, a 10% and a 15% failure rate in the network. Figure 6.6 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.

<sup>2</sup>The model is based on connectivity data gathered from the Mirage testbed.

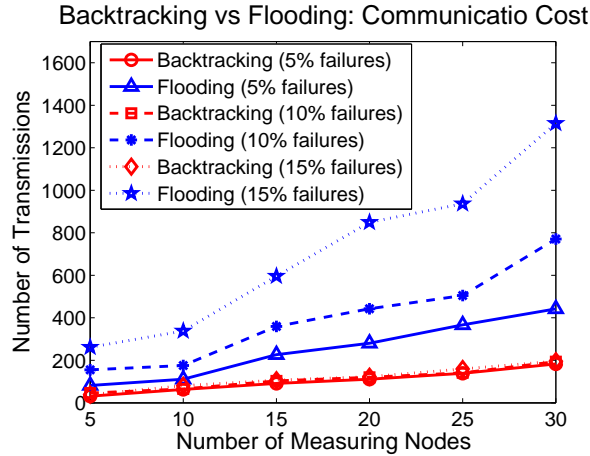


Figure 6.6. Comparison of the 2 recovery algorithms under conditions of failures with rates 5%, 10% and 15% in terms of communication cost. Notice that the backtracking lines practically coincide.

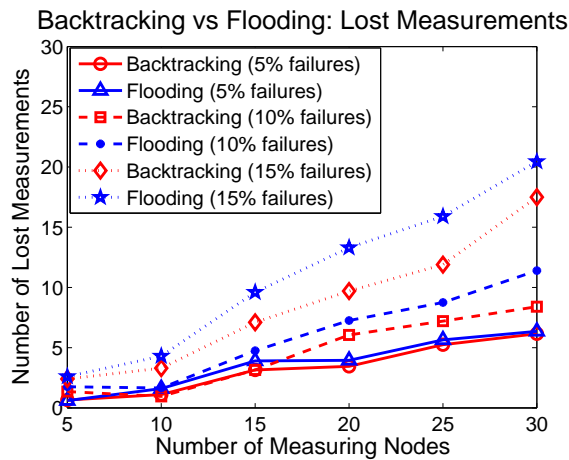


Figure 6.7. Comparison of the 2 recovery algorithms under conditions of failures with rates 5%, 10% and 15% in terms of the number of lost measurements.

### 6.1.2 Discussion

In this section we have presented an evaluation of our heuristics and our recovery algorithms based on Matlab and TOSSIM simulations. All simulations were modeled from traces gathered from a real sensor network testbed.

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 scenar-

ios 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 bad 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 7

# Conclusion

In this work we focused on optimizing the routing paths for data gathering tasks that use source routing. Starting by assuming a semi-static network topology, we defined the optimization problem that we need to solve, and proved that it is NP-hard. We also provided a polynomial time approximation algorithm, for which we proved that the total cost of its solution is bounded by a  $\frac{7}{4}$  constant factor of the optimal cost. We presented the packet structure that is used to inject the routing information into the network, and provided algorithms to adjust the communication paths so that they can be accommodated by any specified packet size. Finally we provided heuristic solutions to recover from failures in the network and presented experimental results on the performance of our algorithms.

In our future work we intend to consider additional recovery methods, and hope to provide theoretical guarantees on the cost of the recovery algorithms and the number of unrecoverable measurements.

## References

- [1] <https://mirage.berkeley.intel-research.net/>.
- [2] 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).
- [3] D. Braginsky and D. Estrin, *Rumor routing algorithm for sensor networks*, ICDCS-22, 2002., 2002.
- [4] David Braginsky and Deborah Estrin, *Rumor routing algorithm for sensor networks*, 1st ACM international workshop on Wireless sensor networks and applications, ACM Press, 2002.
- [5] Moses Charikar, Samir Khuller, and Balaji Raghavachari, *Algorithms for capacitated vehicle routing*, 30th annual ACM symposium on Theory of computing, ACM Press, 1998.
- [6] N. Christofides, *Worst case analysis of a new heuristic for the traveling salesman problem*, Tech. report, Carnegie Mellon University, 1976.
- [7] R. Cristescu, B. Beferull-Lozano, and M. Vetterli, *On network correlated data gathering*, 2004.
- [8] 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.
- [9] 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.
- [10] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong, *Model-driven data acquisition in sensor networks*, VLDB, 2004.
- [11] Deepak Ganesan, Ben Greenstein, Denis Perelyubskiy, Deborah Estrin, and John Heidemann, *Multi-resolution storage and search in sensor networks*, ACM Transactions on Storage (to appear) (2005).
- [12] M. R. Garey and D. S. Johnson, *Computers and intractability – a guide to the theory of np-completeness*, Freeman, San Francisco, 1979.
- [13] M. Haimovich and A. H. G. Rinnooy Kan, *Bounds and heuristics for capacitated routing problems*, Mathematics of Operations Research (1985).
- [14] 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.
- [15] S. T. Hedetniemi, S. M. Hedetniemi, and A. Liestman, *A survey of gossiping and broadcasting in communication networks*, Networks (1998).
- [16] W. Heinzelman, J. Kulik, and H. Balakrishnan, *Adaptive protocols for information dissemination in wireless sensor networks*, 1999.
- [17] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin, *Directed diffusion: a scalable and robust communication paradigm for sensor networks*, 6th annual international conference on Mobile computing and networking, 2000.
- [18] Joanna Kulik, Wendi R. Heinzelman, and Hari Balakrishnan, *Negotiation-based protocols for disseminating information in wireless sensor networks*, Wireless Networks (2002).
- [19] 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.
- [20] 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.
- [21] 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.
- [22] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, *The design of an acquisitional query processor for sensor networks*, ACM SIGMOD, 2003.
- [23] S. Madden and J. Gehrke, *Query processing in sensor networks*, Pervasive Computing **3** (2004), no. 1.
- [24] T. Ralphs, L. Kopman, W. Pulleyblank, and L. Trotter, *The capacitated vehicle routing problem*.
- [25] Narayanan Sadagopan and Bhaskar Krishnamachari, *Maximizing data extraction in energy-limited sensor networks.*, INFOCOM, 2004.
- [26] Anna Scaglione and Sergio D. Servetto, *On the interdependence of routing and data compression in multi-hop sensor networks*, MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking, 2002, pp. 140–147.

- [27] Alec Woo, Terence Tong, and David Culler, *Taming the underlying challenges of reliable multihop routing in sensor networks*, SenSys '03, 2003, pp. 14–27.
- [28] Yang Yu, Bhaskar Krishnamachari, and Viktor K. Prasanna, *Energy-latency tradeoffs for data gathering in wireless sensor networks*, INFOCOM, 2004.