MINIMUM LATENCY AGGREGATION CONVERGECAST IN WIRELESS SENSOR NETWORKS

Jonathan Gagnon

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 2014

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Jonathan Gagnon**

Entitled: **Minimum Latency Aggregation Convergecast in Wireless Sensor Networks**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair
Dr. Olga Ormandjieva

_____ Examiner
Dr. Hovhannes Harutyunyan

_____ Examiner
Dr. Jaroslav Opatrny

_____ Supervisor
Dr. Lata Narayanan

Approved by _____
Chair of Department or Graduate Program Director

_____
Dean of Faculty

Date _____

Abstract

Minimum Latency Aggregation Convergecast in Wireless Sensor Networks

Jonathan Gagnon

In wireless sensor networks, sensor nodes are used to collect data from the environment and send it to a data collection point or a *sink node* using a convergecast tree. Considerable savings in energy can be obtained by *aggregating* data at intermediate nodes along the way to the sink.

We study the problem of finding a minimum latency aggregation tree and transmission schedule in wireless sensor networks. This problem is referred to as Minimum Latency Aggregation Scheduling (MLAS) in the literature and has been proven to be NP-Complete even for unit disk graphs. We present a new simpler proof of the NP-Completeness of the MLAS Problem for arbitrary networks and unit disk graphs. We give tight bounds for the latency of aggregation convergecast for grids, tori, and trees. For regular unit interval graphs, we provide an algorithm which is guaranteed to have a latency that is within one time slot of the optimal latency. Finally, for unit interval graphs we give a 2-approximation algorithm to solve the same problem.

For arbitrary graphs, we introduce a new algorithm for building an aggregation tree. Furthermore, we propose two new approaches for building a transmission schedule to perform aggregation on a given tree. We evaluate the performance of our algorithms through extensive simulations on randomly generated graphs and we compare them to the previous state of the art. Our results show that one of our algorithms has a latency that is 38% less than the latency of the previous best algorithm.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Dr. Lata Narayanan, for her help and guidance throughout this thesis. Her deep knowledge of the field and her valuable suggestions helped me achieve the results presented in this thesis. I have learned a lot from working with her and she gave me the tools that I needed to complete this work. Her availability outside of normal working hours was also very much appreciated.

I also wish to thank my wife and two daughters, who have let me spend the necessary hours to be able to work on this problem. Without their encouragements and support, it would have been very difficult to complete this thesis.

Contents

List of Figures viii

List of Tables xi

1 Introduction 1

1.1 Model and Problem Statement 3

1.2 Notation 6

1.3 Summary of Contributions 7

1.4 Outline of Thesis 7

2 Related Work 9

2.1 Aggregation Techniques for data gathering 10

2.1.1 SPIN family of protocols 11

2.1.2 Directed Diffusion 12

2.1.3 LEACH 13

2.1.4 LEACH-C 14

2.1.5 CLUstered Diffusion with Dynamic Data Aggregation (CLUDDA) 15

2.1.6 Other Approaches 16

2.2 Scheduling Algorithms for Convergecast 17

2.2.1 Optimal and near-optimal algorithms for line and tree networks 18

2.2.2 BFS-Based Approximation Algorithm 21
2.3 Minimum Latency Aggregation Scheduling 22
2.3.1 Shortest Data Aggregation (SDA) 23
2.3.2 Randomized Distributed Algorithm 24
2.3.3 MIS-based Algorithms 25
2.3.4 BSPT-WIRES 26
2.4 Differences with our work 28

3 Optimal and Approximation Algorithms for Specific Topologies 30
3.1 NP-Completeness Proof 30
3.2 Trees 37
3.3 Grids and Tori 40
3.4 Unit Interval Graphs 46
3.4.1 Lower Bound for Unit Interval Graphs 46
3.4.2 Algorithm for Unit Interval Graphs 50
3.4.3 Lower Bound for Regular Unit Interval Graphs 51
3.4.4 Algorithm for Regular Unit Interval Graphs 56

4 Heuristics for Arbitrary Graphs 64
4.1 Degree-Constrained Aggregation Tree (DCAT) 64
4.2 Scheduling Algorithms 67
4.3 Simulation Results for DCAT 71
4.3.1 Performance Comparison for Small Graphs (5 by 5) 72
4.3.2 Performance Comparison for Medium-Sized Graphs (10 by 10) 74
4.3.3 Performance Comparison for Large Graphs (20 by 20) 77
4.4 Simulation Results for Scheduling Algorithms 79
4.4.1 Performance Comparison for Small Graphs (5 by 5) 79

4.4.2 Performance Comparison for Medium-Sized Graphs (10 by 10) 82
4.4.3 Performance Comparison for Large Graphs (20 by 20) 84
4.5 Performance Analysis 88
4.5.1 DCAT Performance 88
4.5.2 WIRES-G and DCATS Performance 92

5 Conclusion and Future Work **96**

Bibliography **98**

List of Figures

1 A multi-line network. 20

2 Reduction of a tree network into linear branches. 20

3 Illustration of a tree and schedule for an instance of 3-SAT. 33

4 Example tree and schedule for a specific instance of 3-SAT. 34

5 Example of an optimal schedule for a perfect binary tree. 38

6 Example of an optimal schedule for a perfect ternary tree. 38

7 Example of a schedule built by the ScheduleGrid algorithm for a 5×5 grid with the sink node located at $(3, 3)$. 43

8 Illustration of a graph with 3 cliques 48

9 Illustration of a schedule built by the Hub Algorithm. 50

10 Optimal solution using the Hub Algorithm. 57

11 Illustration of the tree and schedule built by the Hub Algorithm for a Regular Unit Interval Graph of size $2k$. 57

12 Illustration of using S_0 as a data aggregator for a Regular Unit Interval Graph of size $2k$. 58

13 Tree and schedule for G_1 and G_2. 58

14 Tree and schedule for G_2 and G_3. 59

15 Example showing that minimizing the highest degree in the aggregation tree is not necessarily a good approach. 67

16 Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 5 by 5. 72
17 Average gains of using the DCAT algorithm when compared to the BSPT algorithm for a network size of 5 by 5. 74
18 Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 10 by 10. 75
19 Average gains of using the DCAT algorithm when compared to the BSPT algorithm for a network size of 10 by 10. 76
20 Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 20 by 20. 77
21 Average gains of using the DCAT algorithm when compared to the BSPT algorithm for a network size of 20 by 20. 79
22 Average aggregation convergecast latency for a network size of 5 by 5. 80
23 Average gains of using DCAT, WIRES-G or DCATS when compared to WIRES-BSPT for a network size of 5 by 5. 81
24 Average aggregation convergecast latency for a network size of 10 by 10. 83
25 Average gains of using DCAT, WIRES-G or DCATS when compared to WIRES-BSPT for a network size of 10 by 10. 84
26 Average aggregation convergecast latency for a network size of 20 by 20. 85
27 Average gains of using DCAT, WIRES-G or DCATS when compared to WIRES-BSPT for a network size of 20 by 20. 87
28 Relationship between the degree in the graph and the average number of children in the aggregation tree for a density of 30. 89
29 Relationship between the degree in the graph and the average number of children in the aggregation tree for a density of 100. 90

30 Number of nodes that have a certain number of children in the aggregation tree (density=30). 90
31 Number of nodes that have a certain number of children in the aggregation tree (density=100). 91
32 Relationship between the degree in the graph and the average number of children in the aggregation tree for a density of 30. 92
33 Relationship between the degree in the graph and the average number of children in the aggregation tree for a density of 100. 93
34 Location of the high-degree nodes in the aggregation tree (density=30). 94
35 Location of the high-degree nodes in the aggregation tree (density=100). 94

List of Tables

1 Example convergecast schedule for a 10-node line network 19

2 Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 5 by 5. 73

3 Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 10 by 10. 76

4 Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 20 by 20. 78

5 Average aggregation convergecast latency for a network size of 5 by 5. 80

6 Average latency gains of using DCAT, WIRES-G or DCATS when compared to BSPT-WIRES for a network size of 5 by 5. 82

7 Average aggregation convergecast latency for a network size of 10 by 10. 83

8 Average latency gains of using DCAT, WIRES-G or DCATS when compared to BSPT-WIRES for a network size of 10 by 10. 85

9 Average aggregation convergecast latency for a network size of 20 by 20. 86

10 Average latency gains of using DCAT, WIRES-G or DCATS when compared to BSPT-WIRES for a network size of 20 by 20. 87

Chapter 1

Introduction

A Wireless Sensor Network (WSN) is a collection of densely deployed sensor nodes that collaborate to monitor physical events or conditions [2]. WSNs have a wide range of applications such as environmental monitoring, target detection and tracking, battlefield surveillance, disaster relief, health and home automation. Sensor nodes are low-cost battery-powered devices that are equipped with a radio transceiver and one or more sensors that can detect aspects of the environment such as temperature, humidity, or light. Sensor nodes are usually deployed in an arbitrary manner inside an area to be monitored. The position of sensor nodes is not always predetermined, which means that the nodes must be able to *self-organize* after being deployed. This versatility is useful for some applications where careful deployment is not possible or when the environmental conditions might affect the network topology. Since transmission ranges are small, communication between nodes is achieved by multi-hop routing, with sensor nodes forwarding packets on behalf of other nodes.

The main function of WSNs is to collect information about the environment in which they are deployed. In most applications, the collected information is sent to a selected node called the *sink*. This communication pattern is called *convergecast* [27]

and has been studied extensively in the context of WSNs. Convergecasting is usually done by building a logical tree rooted at the sink and by routing packets along the tree's edges toward the sink. Properly scheduling the nodes' transmissions is important to avoid possible interference.

There are two other main communication patterns that are used in WSNs [27]: *broadcast* and *local gossip*. The broadcast communication pattern is used when a node in the network, usually the base station, needs to send a message to all the other nodes in the network. For example, reprogramming all the nodes in a sensor network may be achieved using a broadcast operation. Local gossip is used when sensor nodes need to collaborate with their neighbors to detect some events in the environment. For example, local gossip may be a useful primitive in a wildfire or target detection application.

Sensor nodes are powered by small batteries, and in many applications, it is infeasible or very expensive to replace or recharge the battery. Therefore, energy efficiency is an overriding concern in the design of communication protocols for wireless sensor networks. Since the radio is by far the most power-hungry element of a sensor node [11], any reduction in the transmitted data can be translated into energy savings for the sensor node. Even though the processing power of sensor nodes is limited, it is usually sufficient for simple computations. This allows for some processing of the raw data to be done before its transmission, and the cost of this local processing is negligible compared to the cost of transmissions. For example, a sensor node could compress the sensor readings, or send a simple function of the sensor readings, thus reducing the size of the packets it sends. Additionally, in a convergecast operation, a sensor node could *combine* multiple packets received from its children, perhaps with its own data, before forwarding it to its parent in the tree to reduce the number of its

own transmitted packets. Finally, in some applications, the information can be *aggregated* along the way to the sink, using a specific aggregation function. For example, suppose you want to calculate the average temperature or the top-k temperatures in a region. Each node can easily aggregate the data received from its children with its own and simply send one message containing the result. In large networks, this can dramatically reduce the total number and size of packets sent, because each node sends only one message and the total number of messages sent is always equal to $n - 1$.

In this thesis, we study the problem of convergecast with aggregation. The *latency* of a convergecast operation is the time taken for the sink node to receive the data from all the nodes. The problem of minimizing this latency is referred to as the Minimum Latency Aggregation Scheduling (MLAS) problem [37] and has also been called aggregation convergecast [30] and MDAT (Minimum Data Aggregation Time) [8] in the literature.

Algorithms solving the MLAS problem are usually very different from algorithms solving the regular convergecast problem. The main difference is that in the MLAS problem, nodes have to wait until they have performed data aggregation on all incoming packets before they can transmit. Thus, the scheduling of nodes' transmissions in the MLAS problem is different from regular convergecast where a node can simply immediately forward any packet it gets from a child.

1.1 Model and Problem Statement

Throughout this thesis, we assume that the nodes are synchronized and that they share the same wireless channel. Time is assumed to be slotted, and each node is scheduled to transmit in a given slot. Two nodes can transmit in the same time

slot so long as their transmissions do not interfere. All nodes are stationary and their transmission range is assumed to be constant and identical. The interference radius is assumed to be equal to the transmission range. Nodes are allowed to use an aggregation function, as long as the aggregation function can be computed in a distributed fashion by intermediate nodes in the tree and that it requires O(1) amount of information to be forwarded by intermediate nodes. Examples of functions that follow these constraints are the aggregation functions *Min*, *Max*, *Average*, *Sum* and *Count*. Note that the *Median* aggregation function is not included in the list as it requires non-constant amount of information to be forwarded by intermediate nodes.

Given a set of sensor nodes $S = \{S_0, S_1, \ldots, S_{n-1}\}$ with S_{n-1} being the sink node and where each node has a data item that it wants to send to the sink node, the problem we are interested in is to find a transmission schedule to send all the aggregated data to the sink in such a manner that each node transmits exactly once. A valid schedule for the problem has the following constraints [30]:

1. Each node must transmit exactly once, except for the sink which doesn't transmit.

2. A node cannot receive after it has transmitted.

3. A node cannot transmit and receive in the same time slot.

4. When a node is receiving, exactly one of its neighbors in the graph is transmitting in the same time slot.

The fourth constraint ensures that any solution provides a collision-free schedule. As explained in [30], any solution that follows these constraints must be a tree, so any algorithm for the problem must build a tree rooted at the sink.

Formally, given a graph $G = (V, E)$, and a spanning tree T of G rooted at and directed towards the sink node $s \in V$, we define a *valid schedule* for (T, G) to be an assignment $A : V \to \mathbb{Z}$ of time slots to the nodes of the graph such that

1. $v \in children(u) \Longrightarrow A(u) > A(v)$
2. $(u, v) \in T$ and $(w, v) \in G \Longrightarrow A(u) \neq A(w)$

The *latency* of a valid schedule is defined to be $\max_{v \in V}\{A(v)\}$. The MLAS problem is now formally defined as follows: Given a graph $G = (V, E)$, find a spanning tree T of G and a valid schedule of minimum latency for (T, G).

Sometimes sensor nodes can be deployed to monitor perimeters or borders. These sensors will most of the time overlap to provide some kind of redundancy. This kind of network can usually be represented as a unit interval graph where the unit intervals represent the nodes' transmission range (which is assumed to be constant and identical).

We consider a unit interval graph $G = (V, E)$ of size n, where $V = \{S_0, ..., S_{n-1}\}$ and where S_{n-1} is the sink node. We assume that all sensor nodes are located at distinct locations. The nodes are sorted in descending order of distance from the sink which means S_0 is the farthest node from the sink. S_j is called a forward neighbor of S_i if it is closer to the sink than S_i (i.e. if $j > i$), otherwise it is called a backward neighbor.

We also study a more constrained kind of unit interval graph where all nodes except for the last k nodes have k forward neighbors. We call such a graph a *regular unit interval graph*.

1.2 Notation

In this section, we describe the notation used throughout the algorithms presented in this thesis. Variables used in the pseudocode are considered to be objects that can have attributes. For example, a graph G contains vertices and edges which are referred to as $G.V$ and $G.E$ respectively. Consider a vertex v, here is the list of attributes it can have:

1. $v.d$ denotes the distance in hops between v and the sink.
2. $v.t$ denotes the time slot assigned to v to transmit its aggregated data to its parent in the aggregation tree.
3. $v.p$ denotes the parent of v in the aggregation tree. It can be a pointer to any neighbor in G or nil if v has not been assigned a parent.

We denote by $\mathcal{N}(v)$ the set of neighbors of v and we denote by $\mathcal{N}(S)$ the set of nodes that are a neighbor of at least one node in the set S. Finally, we denote by $\mathcal{C}(v)$ the children of v in the aggregation tree and by $\mathcal{C}_i(v)$ the ith child of v in the aggregation tree.

1.3 Summary of Contributions

The MLAS problem has been shown to be NP-Complete even for unit disk graphs by Chen et al. [8]. We present a new proof for the NP-Completeness of the MLAS problem for arbitrary networks and unit disk graphs. The transformation used in our proof is simpler than the one in [8] because of the fact that we give a sequence of two reductions.

We prove lower bounds for the latency of aggregation convergecast for grids, tori, and trees and we provide algorithms with matching upper bounds. For regular unit interval graphs, we provide an algorithm which is guaranteed to have a latency that is within one time slot of the optimal latency. For unit interval graphs, we give a 2-approximation algorithm to solve the same problem. Our 2-approximation algorithm compares favorably to the best known approximation ratio of $\triangle - 1$ for general graphs.

For arbitrary graphs, we give a new algorithm for building an aggregation tree. Furthermore, we give two new approaches for building a transmission schedule to perform aggregation on a given tree. We evaluate the performance of our algorithms through extensive simulations on randomly generated graphs and we compare them to the previous state of the art. Our results show that one of our algorithms beats the previous best by up to 38%.

1.4 Outline of Thesis

The remainder of this thesis is organized as follows. In Chapter 2, we present a review of the relevant literature on data gathering in WSNs. Our proof of NP-Completeness and our lower bounds and algorithms for trees, gris, tori and unit interval graphs are presented in Chapter 3. In Chapter 4, we present a new tree-building algorithm and two new scheduling algorithms, and we compare them to the state of the art through

simulations. Finally, conclusions and suggestions for future research on the subject are given in Chapter 5.

Chapter 2

Related Work

Broadcast has been studied extensively in various models of communication for wired networks [17], as well as in wireless networks [9]. In the wired setting, the time for broadcast on a tree is the same as the time for convergecast (sometimes called accumulation or gathering). However, in the wireless setting, these two times are usually quite different. First, the broadcast nature of wireless transmissions implies that a node can reach all its children in a single transmission. Algorithms for broadcast can take advantage of this, while algorithms for convergecast cannot. For example, consider a tree where the root node has two children. Broadcast can be accomplished in one time slot, while two time slots are needed for convergecast. Second, in a wireless network, a valid schedule has to consider interference caused by edges *not in the tree*; there is no such restriction in the wired setting. For example, suppose nodes a and c have parents b and d respectively in the convergecast tree. If node a is scheduled to transmit at time t to its parent b, then node c cannot be scheduled to transmit at time t to its parent d if c happens to be a neighbor of b in the graph or if a happens to be a neighbor of d in the graph. To summarize, convergecast in wireless networks is a different problem than either broadcast in wireless networks or convergecast in

wired networks.

In this chapter, we review existing data gathering algorithms for wireless sensor networks. Although many protocols and algorithms have been proposed for traditional wireless ad hoc networks, it is widely accepted that those algorithms are not well suited for WSNs [3]. One of the main reasons is that sensor nodes are much more limited in power, which means that network protocols need to focus on power conservation, rather than achieving high quality of service (QoS) like traditional network protocols. WSNs are also usually much denser than traditional wireless ad hoc networks and node densities may be as high as 20 nodes/m^3 [33]. This high density comes with challenges because a high number of devices will contend for the same wireless communication channel. To address those constraints, extensive research has been done to devise protocols and algorithms optimized specifically for WSNs.

In Section 2.1, we look at some of the first efforts suggesting the use of application-specific logic to do some form of aggregation in wireless sensor networks. Section 2.2 describes algorithms for the regular convergecast problem where all packets need to be forwarded to the sink and no in-network aggregation takes place. Existing algorithms that generate a schedule for the MLAS problem are explored in Section 2.3. Finally, we explain the difference between our work and what has been done before in Section 2.4.

2.1 Aggregation Techniques for data gathering

In this section, we look at some of the first strategies that were used in order to reduce the amount of traffic in wireless sensor networks. We will show that many protocols use application-specific logic to reduce the number of packets sent towards the sink while others use some form of message-packing to achieve the same goal.

2.1.1 SPIN family of protocols

A family of adaptive protocols called SPIN (Sensor Protocols for Information via Negotiation) [20] was designed to efficiently disseminate information among sensors in an energy-constrained wireless sensor network. The design of SPIN came from the analysis of the limitations of conventional protocols, characterized as *classic flooding* in [20], for disseminating data in a sensor network. Three deficiencies of classic flooding were identified as problems in the context of sensor networks:

1. *Implosion*: Implosion happens when a node receives the same message multiple times from different sources. For example, if a node A has x neighbors that are also the neighbors of another node B, B will receive x copies of the message sent by A.

2. *Overlap*: Because sensor nodes are densely deployed, some nodes may sense the same information at the same time. As a result, energy and bandwidth is wasted by sending the same information more than once.

3. *Resource blindness*: Nodes do not modify their activities based on the amount of energy left.

To overcome the problems of implosion and overlap, the SPIN family of protocols use high-level data descriptors, called *meta-data*, to eliminate redundant data throughout the network. Before sending the actual data, a sensor node first broadcasts the meta-data to all its neighbors by sending a new data advertisement (ADV) message. Neighbors that are interested in the data respond with a request for data message (REQ). The data message (DATA) is then only sent to interested neighbors. This approach is called *data-centric* in that all communication is for named data[22]. In the experiments done in [20], this three-stage approach effectively cuts down on

wasted energy due to redundant information. The problem of resource blindness is handled by making the SPIN protocols resource-aware. If resources are low, a node is able to cut back on certain activities to increase longevity. There are two main protocols in the SPIN family: SPIN-1 and SPIN-2 [4]. The SPIN-1 protocol implements the three-stage approach described earlier (ADV, REQ, DATA). SPIN-2 extends SPIN-1 by incorporating a threshold-based resource awareness mechanism. Other protocols of the SPIN family are[4, 26]:

1. SPIN-BC: A three-stage handshake protocol designed for broadcast channels.
2. SPIN-PP: A three-stage handshake protocol for point-to-point communication.
3. SPIN-EC: Same as SPIN-PP, but with a low-energy threshold.
4. SPIN-RL: A reliable version of SPIN-BC for lossy networks.

2.1.2 Directed Diffusion

Intanagonwiwat et al. [22] introduced a new data dissemination paradigm called *directed diffusion*. Like the SPIN protocols, directed diffusion is data-centric in that all communication is for named data. However, instead of the sensor nodes advertising information about available data, the sink sends out interest to all sensors by sending a task description, and data matching that task description is then routed toward the sink. An interesting property of directed diffusion is that it allows for multiple sinks to be present in the network with each of them having possibly different interests. Data is aggregated by intermediate nodes to reduce the number of duplicate messages, but duplicates are not removed entirely and some form of redundancy is kept. This allows the protocol to be robust in the presence of node failures. In [23], experiments with a node failure rate of 10% to 20% showed that the event delivery

rate remains high and that the additional delay incurred by failed transmissions is always less than 20%.

2.1.3 LEACH

Heinzelman et al. [18] proposed a clustering-based protocol called LEACH (Low-Energy Adaptive Clustering Hierarchy), that utilizes randomized rotation of local cluster-heads to evenly distribute the energy load among the sensors in the network. The LEACH protocol incorporates data aggregation (also called *data fusion* in [18]) into the routing protocol to reduce the amount of information that must be transmitted to the sink.

In LEACH, each round has a set-up phase where clusters are formed, followed by a steady-state phase where the data collected by the sensors is routed to the sink. In the set-up phase, each node decides whether or not to become a cluster-head by choosing a random number between 0 and 1. If this random number is less than a threshold $T(v)$, the node becomes a cluster-head. The threshold is calculated using the following formula:

$$T(v) = \begin{cases} \frac{P}{1-P[r \mod (1/P)]} & \text{if } v \in G \\ 0 & \text{otherwise} \end{cases}$$

where P is the desired percentage of cluster heads in the network, r is the current round number and G is the set of nodes eligible to become a cluster-head. A node is guaranteed to become a cluster-head within $\frac{1}{P}$ rounds and nodes are removed from the eligible nodes G when they become a cluster-head. After $\frac{1}{P}$ rounds, the value of r is reset to 0 and all nodes become eligible again to be a cluster-head.

Newly elected cluster-heads need to advertise their new state to the other sensor nodes. Each sensor node selects as its parent the cluster-head that has the highest signal strength. Once a parent is selected, the node informs the cluster-head that

it will be a member of the cluster. Afterwards, each cluster-head creates a schedule to determine the time slot in which each node in the cluster will transmit, and the schedule is broadcasted to all the nodes in the cluster.

During the steady-state phase, sensor nodes collect data from their environment and wait for their assigned time slot to send it to their cluster-head. The cluster-head needs to keep its radio on to receive the data from all the nodes in the cluster, but the other nodes in the cluster can shut down their radio to save power. Once the cluster-head has received data from all the nodes in the cluster, it sends the aggregated data to the sink. Since the sink can be far away, this transmission requires a significant amount of energy and assumes that the sensor radio is powerful enough to reach the sink. The steady-state phase goes on for a pre-determined amount of time without changing the clusters. The steady-state phase is usually much longer than the set-up phase to minimize the set-up overhead. The exact time of each phase need to be determined in advance to maintain the synchronization between all the nodes in the network throughout all the rounds. When the steady-state phase is completed, a new round begins, new cluster-heads are selected and new clusters are formed.

Software simulations were conducted in [18] to show that LEACH can reduce energy consumption by as much as a factor of 8 compared with conventional routing protocols. In the same tests, LEACH was able to double the useful network lifetime by evenly distributing the task of aggregating the data from the nodes in the cluster and sending it to the sink.

2.1.4 LEACH-C

Although the LEACH protocol obtains good results compared to previous techniques, the distributed nature of the algorithm means that clusters might not be set up optimally. A centralized version of LEACH, called LEACH-C was proposed in [19]

to try to optimize the cluster formation. Only the nodes that have an energy level above the average energy level in the network are considered in the cluster head selection. Since the problem of finding k optimal clusters is known to be NP-Hard, the simulated annealing algorithm is used to try to efficiently form clusters. In their simulations, LEACH-C is up to 40% more energy-efficient than LEACH because of the improved cluster setup.

Even though the LEACH and LEACH-C protocols improve the energy-efficiency and the lifetime of the network, they both suffer from the same limitation. These protocols rely on the capability of every sensor in the network to communicate directly with the sink node in a single hop. This is not realistic for most WSN where sensor nodes have limited energy and transmission power.

The SPIN and LEACH protocols and the directed diffusion paradigm were important advances in WSNs research, because they showed that there were advantages in application-specific optimizations in the context of sensor networks [23]. SPIN uses application-specific metadata to significantly reduce energy dissipation, whereas LEACH is able to achieve energy savings by doing application-specific data aggregation at its cluster heads and directed diffusion is able to do the same at any intermediate node between the source and the sink.

2.1.5 CLUstered Diffusion with Dynamic Data Aggregation (CLUDDA)

An hybrid approach that combines clustering with directed diffusion was proposed by Chatterjea and Havinga [7]. The approach is called CLUstered Diffusion with Dynamic Data Aggregation (CLUDDA) and it works by using clustering in the initial phase of query propagation of directed diffusion. Instead of communicating directly with the sink, cluster heads communicate between each other in a multi-hop fashion

to route messages to and from the sink. This has the potential of reducing the energy consumption when compared to clustering-only solutions like LEACH. Yet, as mentioned in [32], CLUDDA would need to be implemented and compared with other approaches to validate the potential improvements. The authors of [32] expressed doubts regarding the feasibility of the algorithm for sensor networks because of the potentially high memory requirements at the cluster heads.

2.1.6 Other Approaches

Instead of proposing another application-specific aggregation scheme, He et al. [16] proposed an application-independent data aggregation scheme called AIDA (Application Independent Data Aggregation). The main idea behind AIDA is to maximize the utilization of the communication channel by combining multiple messages into a single packet. It comes from the observation that there is a lot of control overhead in wireless communications and that sending one big packet is more efficient than sending several smaller ones containing the same data. Experimental results show that AIDA reduces the latency by up to 80% and the energy spent by 30-50% under heavy traffic when compared to not using aggregation. Moreover, AIDA can easily be combined with any application-specific aggregation scheme to achieve even better results.

The TAG service, proposed by [29], is a service implemented on top of TinyOS [1] that allows users to send queries to the network. Sensor nodes process the query and route the information back to the base station, aggregating it using the aggregation function specified in the query. An SQL-like language is used to express the queries sent by the users. A more efficient approach for top-k aggregation functions is presented in [38] and called FILA. The main idea behind the FILA approach is to avoid sending readings entirely if they are within certain bounds. The base station

sets up those filters per node according to the data it received from each of them. As long as the data sampled by the sensor node is within the values excluded by the filter, the node refrains from sending a message. For the top-k nodes, another tighter filter is maintained and used to make the top-k report any changes in their readings that are outside the desired tolerance. If the reading is within the tolerance for a top-k node, it doesn't have to communicate the value to the base station and the value is assumed to be the same. Obviously, this filter-based approach is useful when readings are relatively stable from one round to the other. For applications where the readings are more volatile, the base station would keep sending new filters to the nodes and a lot a energy would be lost in these updates. Experiments were done in [38] to compare the FILA approach to TAG [29] and range caching [31], using real world temperature and dew point readings. The results demonstrated that FILA was the best approach for aggregating the top-k values for this type of readings.

2.2 Scheduling Algorithms for Convergecast

In this section, we look at algorithms that build a schedule for the regular convergecast problem (i.e. without aggregation). As stated earlier, convergecast is a communication pattern where the data collected at each node in the network is routed toward a special node in the network called the sink node. Nodes can periodically send data to the sink based on a desired sampling rate or they can respond to a query from the sink node, broadcasted to all the sensor nodes.

When nodes respond to a query from the sink node, the broadcast tree used for sending the query from the sink to the nodes can be used to gather the information from the nodes to the sink. As shown in [5], traditional broadcast algorithms usually don't produce trees that are efficient for the convergecast operation in WSNs. For

applications that require both broadcasting and convergecasting, they proposed a new heuristic algorithm called Convergecasting Tree Construction and Channel Allocation Algorithm (CTCCAA). It was later shown by Upadhyayula et al. [35] that this algorithm was inefficient in terms of energy consumption and latency, and a new more efficient algorithm was proposed in [35].

The problem of minimizing the latency of the convergecast communication in WSNs has been proven to be NP-hard for general graphs by Choi et al. [10] and later by Ergen and Varaiya [12].

2.2.1 Optimal and near-optimal algorithms for line and tree networks

The convergecast problem was solved optimally by Choi et al. [10] for line and tree networks, using pipeline-like scheduling algorithms called LPIPE and TPIPE respectively. The LPIPE algorithm works by scheduling a maximum of 1 out of 3 nodes at every round, which is the maximum that can be scheduled without conflicts when all nodes are transmitting toward the sink. It is shown in [10] that LPIPE creates a schedule that uses exactly $3(n-2)$ time slots for $n > 2$, and that this is an optimal solution. Later and seemingly independently, Gandham et al. [13] proved that an optimal solution for the same line network required a minimum of $3(n-2)+2+1 = 3n-3$ time slots, which appears to be 3 more time slots than what was found by Choi et al. [10]. However, their definition of n is different from the one in [10] in that the sink is not included in n, which accounts for the added three time slots in the lower bound. Table 1 shows an example schedule for a 10-node line network.

The TPIPE algorithm is similar to the LPIPE algorithm and produces schedules that use exactly $3(n-2)$ time slots. However, the algorithm assumes that the root node has only one child and that this child also has only one child, which somewhat

time \ edge	(1,0)	(2,1)	(3,2)	(4,3)	(5,4)	(6,5)	(7,6)	(8,7)	(9,8)
1	T(1)			T(4)			T(7)		
2		T(2)			T(5)			T(8)	
3			T(3)			T(6)			T(9)
4	T(2)			T(5)			T(8)		
5		T(3)			T(6)			T(9)	
6			T(4)			T(7)			
7	T(3)			T(6)			T(9)		
8		T(4)			T(7)				
9			T(5)			T(8)			
10	T(4)			T(7)					
11		T(5)			T(8)				
12			T(6)			T(9)			
13	T(5)			T(8)					
14		T(6)			T(9)				
15			T(7)						
16	T(6)			T(9)					
17		T(7)							
18			T(8)						
19	T(7)								
20		T(8)							
21			T(9)						
22	T(8)								
23		T(9)							
24	T(9)								

Table 1: Example convergecast schedule for a 10-node line network where node 0 is the sink (reproduced from Fig.3 in [10]). To avoid collisions, no more than a third of the nodes can transmit at any time slot.

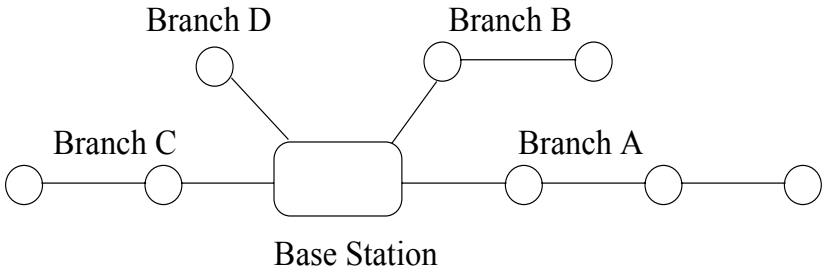

Figure 1: A multi-line network (reproduction of Fig. 3 in [13]).

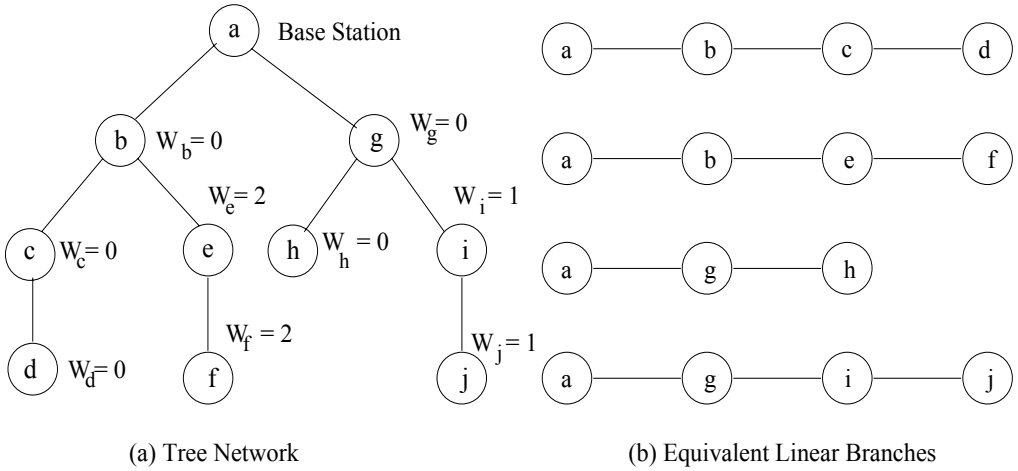

Figure 2: Reduction of a tree network into linear branches (reproduction of Fig. 5 in [13]).

limits the usefulness of the algorithm and mitigates their claim that their TPIPE algorithm produces an optimal schedule for tree networks. Gandham et al. [13] later presented a distributed algorithm for the tree topology without any limitations about the nodes' degree. Their tree algorithm is an adaptation of their multi-line (see Figure 1) algorithm and comes from the observation that a tree network can be reduced to a multi-line network with each line represented as a combination of linear branches of nodes (see Figure 2).

The main idea behind their tree and multi-line algorithms is to schedule transmissions in parallel along multiple branches or subtrees. The bottleneck is usually located at the sink because the sink can only receive one packet at a time. Thus, in each timeslot, the algorithm will need to decide which branch or *one-hop-subtree* will

forward a packet to the sink. The term one-hop-subtree refers to any subtree that is rooted at a one-hop neighbor of the sink. The tree and multi-line algorithms produce schedules that require at most $max(3n_k - 1, n)$ time slots, where n represents the number of nodes in the network and n_k represents the number of nodes in the largest one-hop-subtree or branch. This is a maximum of 2 time slots above the proven lower bound of $max(3n_k - 3, n)$ [13].

2.2.2 BFS-Based Approximation Algorithm

Gandham et al. [13] also proposed a distributed convergecast algorithm that requires at most $3n$ time slots in arbitrary networks. The algorithm works by creating a Breadth First Search (BFS) tree and then using their tree scheduling algorithm to build the schedule. They demonstrated through simulations that this algorithm produces schedules that use an average of around $1.5n$ time slots. In [14], they showed that the same algorithm can be used when nodes generate multiple packets, where P represents the number of packets that need to be sent to the sink. In this scenario, the algorithm produces schedules that use at most $3P$ time slots.

More recently, Augustine et al. [6] looked at variation of the convergecast problem where k readings from sensors can be packed into the same packet for the same energy cost. They refer to this problem as the CCP (ConvergeCast Problem) and they study a simplification of this problem, called the UCCP (Unit ConvergeCast Problem), where the size of each reading is exactly 1 byte. They prove that both CCP and UCCP are NP-Hard and they propose an optimal algorithm for the tree topology and an asymptotically optimal solution for the grid topology.

2.3 Minimum Latency Aggregation Scheduling

Data aggregation has been proposed early on in WSNs to reduce the energy usage of sensor nodes and improve the network lifetime. Krishnamachari et al. [25] demonstrated that significant energy savings could be achieved by using data aggregation, and that the gains were even bigger for large networks or for cases where sources are far from the sink. As mentioned earlier, there are many different forms of data aggregation. Some approaches are application-independent like AIDA [16] and they usually work by trying to pack many readings into a single message. Other approaches work by using an application-specific aggregation function to fuse the data and reduce the number of messages. Such aggregation functions include duplicates removal, min, max, count, average, top-k, or any other function that takes multiple inputs.

In this thesis, we look at the problem of convergecast with in-network data aggregation using an aggregation function that produces a single output, regardless of the number of inputs. The problem of minimizing the time it takes to aggregate the information from all the nodes in the network using such an aggregation function is referred to as MLAS (Minimum Latency Aggregation Scheduling) [37]. The problem has also been variably called aggregation convergecast [30] and MDAT (Minimum Data Aggregation Time) [8] in the literature.

Algorithms that solve the MLAS problem are divided into two main categories: centralized and distributed algorithms. Centralized algorithms usually generate better schedules as they have complete knowledge of the network, but they have the disadvantage of being less adaptable to topology changes that could occur due to node failures. Distributed algorithms are better suited for environments in which topology changes occur frequently, as they don't need to wait for a new schedule to be broadcasted to all the nodes. However, they usually have a higher latency than centralized algorithms. Most centralized algorithms solving the MLAS problem are

divided into 2 phases. The first phase builds a logical tree rooted at the sink and the second phase schedules the transmissions along that tree.

In this section, we describe several centralized and distributed algorithms that solve the MLAS problem. All these algorithms solve the problem for arbitrary network topologies and have proven latency bounds.

2.3.1 Shortest Data Aggregation (SDA)

Chen et al. [8] introduced a centralized $(\triangle - 1)$-approximation algorithm named Shortest Data Aggregation (SDA), where $\triangle$ is the maximum degree. The algorithm works by first creating a spanning tree T_1 rooted at the sink. At each iteration, the algorithm selects some nodes to transmit for the round corresponding to the iteration, and then those nodes are removed from the tree. For instance, suppose that T_r is the tree to schedule at the r^{th} iteration. The algorithm iterates through all the leaves in T_r in decreasing order of the number of their non-leaf neighbors in T_r. A leaf is eliminated from the potential senders if it can be eliminated without violating the property that every non-leaf neighbor of a leaf in T_r has a neighbor in the remaining potential senders. At the end of the loop, the leaves that have not been eliminated are scheduled to transmit at time r. For each scheduled node, the receiver is selected from the non-leaf neighbors of the node in T_r and the tree T_{r+1} is built by removing those leaves from T_r. The algorithm terminates when all nodes but the sink have been scheduled.

As pointed out in [30], the fact that the scheduling part of SDA selects a receiver instead of just using the parent node in T_r means that the original aggregation tree is not always kept. Because of that, the authors of [30] argue that the SDA algorithm cannot be used when a given aggregation tree is required for a specific application.

2.3.2 Randomized Distributed Algorithm

A randomized distributed algorithm called DC (Distributed Convergecast) was proposed by Kesselman and Kowalski [24]. The DC algorithm works by dividing the time into rounds long enough for the transmission of one message. At the beginning of the first round, all nodes are active and have data to transmit. At each round, a subset of active nodes are randomly selected to transmit. The transmission range of each selected node is set to the distance between the sender and the closest active neighbor. If the transmission is successful (i.e. no collision detected), the sender becomes inactive. The senders for which the transmission failed remain active and can be selected or not to transmit in the next round. The nodes that received data from another node then fuse it with their own data. This process goes on until only one node is active. At this point, the remaining node has aggregated the data from all the nodes in the network.

There are a few issues with this algorithm. As pointed out by other authors [41, 40], the algorithm assumes that sensor nodes can easily adjust their transmission range and that they have a special collision detection capability. This is usually not the case for the hardware present in WSNs since the cost must remain low because of the large number of nodes deployed. Moreover, they assume that the maximum transmission range is unbounded, which is not very realistic for large networks. In their own evaluation of the algorithm, they acknowledge the fact that the algorithm may require $O(n \log n)$ times the minimum energy required to complete the convergecast. Last but not least, because of the random nature of the algorithm, the sink node changes at every data collection. This makes it impractical if the data needs to be collected at a predetermined node.

2.3.3 MIS-based Algorithms

Huang et al. [21] designed a centralized algorithm based on Maximal Independent Sets (MIS) and with a latency bound of $23R + \triangle - 18$, where R is the maximum distance between the sink and any other node. Their main improvement over SDA is that the maximum degree contributes to an additive factor instead of a multiplicative factor in the latency bound. However, the algorithm doesn't perform well for a big enough R, and it has been found that the generated schedule is not always conflict-free[40].

Using a similar MIS approach, Wan et al. [37] proposed three new centralized algorithms, SAS, PAS and E-PAS, with latency bounds of $15R+\triangle-4$, $2R+O(\log R)+\triangle$ and $(1+O(\log R/\sqrt[3]{R}))R+\triangle$ respectively. The E-PAS algorithms has the lowest proved maximal latency, although experiments have shown that approaches based on the Connected Dominating Set (CDS) problem don't perform well in practice [30]. Indeed, CDS and MIS approaches tend to connect many nodes to the same parent which prevents parallelism and increases latency.

Yu et al. [40] proposed a distributed MIS-based algorithm called DAS (Distributed Aggregation Scheduling). Their algorithm starts by building an aggregation tree using the distributed approach of constructing a CDS tree proposed by Wan et al. [36]. To build the schedule in a distributed manner, each node needs to know its unique ID and maintains a list of nodes called the *competitors*. For a given node u, the competitors of u are all the nodes that cannot be scheduled at the same time as u. This list of competitors is used to build a conflict-free schedule. Nodes that have a higher ID than all their competitors are scheduled to transmit first. Then the other nodes are scheduled as soon as all their competitors with higher IDs have transmitted. At any time slot, a node can be in any one of six possible states and messages are sent between the nodes to be able to properly schedule the nodes in a distributed

way. DAS has a proven maximum latency of $24D + 6\triangle + 16$, where D is the diameter of the network.

Another distributed MIS-based algorithm, also called DAS (Data Aggregation Scheduling), is proposed by Xu et al. [39]. Their algorithm was also inspired by the distributed approach of constructing a CDS proposed in [36]. Their DAS is a little different than previous CDS-based algorithms in that data is first aggregated towards the *center* of the network and then routed to the sink. The network center is defined as the node that minimizes the maximum distance between itself and any other node in the network. By selecting the center as the aggregation point, the theoretical upper bound of the algorithm is reduced. They prove an upper bound of $16R + \triangle - 14$ where they define R as the maximum distance between the network center and any node instead of being the maximum distance between the sink and any node.

2.3.4 BSPT-WIRES

Malhotra et al. [30] introduced two centralized algorithms, one is a tree construction algorithm called BSPT (Balanced Shortest Path Tree) and the other is a ranking/priority-based scheduling algorithm called WIRES (Weighted Incremental Ranking for convergEcast with aggregation Scheduling). We present this work in more details since in Chapter 4 we compare our algorithms experimentally with their work. The BSPT algorithm is based on a lower bound that they introduced for the latency of the aggregation convergecast in a logical tree, and given by the following theorem (Theorem 1 in [30]) :

Theorem 1 *Given a logical tree the lower bound, T_{min}, for the aggregation convergecast scheduling problem is $max\{\mathcal{E}_i + h_i : i = 1, 2, \ldots, n\}$, where $\mathcal{E}_i$ and h_i, respectively, are the children-count and hop-count (from the root) of node i in the given tree.*

The BSPT algorithm builds a shortest path tree by traversing the graph in a Breadth First Search (BFS) way. As it builds the tree, the BSPT algorithm builds a bipartite graph for all the nodes of two consecutive levels in the tree, and it uses the bipartite semi-matching algorithm from Harvey et al. [15] to distribute as evenly as possible the lower level nodes between the upper level nodes. This is done for every pair of consecutive levels in the tree. The result is a shortest path tree that minimizes the lower bound presented in Theorem 1, compared to any other possible shortest path trees generated for the same graph.

Algorithm 1 WIRES

Input: $G = (V, E)$, s: sink node, $v.p$: parent of $v \in V$ in the tree
Output: A valid schedule for G where $v.t$ is the transmission time for $v \in V$

1: **procedure** WIRES(G, s)
2: $\forall v \in G.V \ v.t = 0$ ▷ Initialize time slots
3: $j = 1$
4: **while** $s.t = 0$ **do**
5: $L =$ GetEligibleNodes(G)
6: ComputeWeights(L)
7: SortDecreasing(L) ▷ Sort nodes in decreasing order of weights.
8: $S = R = \emptyset$ ▷ Set of senders and receivers are initially empty
9: ScheduleNodes(L, S, R)
10: $j = j + 1$
11: **end while**
12: **end procedure**

13: **procedure** ScheduleNodes(L, S, R)
14: **for each** $u \in L$ **do**
15: **if** $u \notin \mathcal{N}(R)$ and $u.p \notin \mathcal{N}(S)$ **then** ▷ If u can transmit without conflict
16: $u.t = j$ ▷ u is scheduled to transmit at time j
17: $S = S \cup \{u\}$ ▷ u is added to the set of senders
18: $R = R \cup \{u.p\}$ ▷ The parent of u is added to the set of receivers
19: **end if**
20: **end for**
21: **end procedure**

The WIRES scheduling algorithm takes an aggregation tree as input and starts by considering all the leaves in the tree as nodes eligible to be scheduled at time 1.

A weight is calculated for every eligible node where a higher weight means a higher priority for the node to be scheduled at the current time slot. Eligible nodes are then considered one by one and each node that can transmit without interfering with the previous nodes is scheduled. Once all nodes have been considered, the round is completed and the set of eligible nodes is updated by removing the nodes that have been scheduled, and by adding the nodes that have received data from all their children. The previous steps are repeated until all the nodes have a schedule.

The weight computation that worked best in their experiments was to use the *non-leaf neighbor count*, which corresponds to the number of neighbors that are not leaves in the aggregation tree. Since scheduled nodes are removed from the aggregation tree once they are scheduled, the weights need to be recomputed at every round. Experiments showed that the combination of BSPT and WIRES was better than SDA, SAS, PAS and DAS [40] by 10% to 30%. They also proved that the choice of a good aggregation tree was very important by combining their BSPT algorithm with the scheduling algorithms of SDA, PAS and DAS. They called those modified algorithms SDA-BSPT, PAS-BSPT and DAS-BSPT respectively, and their experiments showed that the latency of the schedules produced by those algorithms was reduced when compared to their original versions. The WIRES-BSPT algorithm still has an edge of around 10% to 18% in most cases. The only exception is at very high densities, where the SDA-BSPT algorithm was able to slightly outperform their own WIRES-BSPT.

2.4 Differences with our work

Previous work on the MLAS problem, presented in Section 2.3, focused mainly on finding algorithms that work for any topology. Some papers introduced algorithms

with provable latency bounds while others presented practical algorithms that were evaluated through simulations.

To the best of our knowledge, no paper has addressed the MLAS problem for specific simple topologies like trees, grids, tori and unit interval graphs. In our work, we present new lower bounds for all these topologies, and we present optimal algorithms for trees, grids and tori as well as near-optimal algorithms for unit interval graphs and regular unit interval graphs.

We propose a new algorithm to construct the logical convergecast tree used to gather information to the sink. When used in combination with the WIRES [30] scheduling algorithm, we show through simulations that our algorithm beats the BSPT-WIRES combination for random topologies. According to simulations done in [30], BSPT-WIRES was the previous best combination of algorithms for solving the MLAS problem. We propose two new scheduling algorithms, WIRES-G and DCATS, that both produce schedules with lower latencies than WIRES.

We also present an NP-Completeness proof for the general MLAS problem and we extend this proof to show that the problem is also NP-Complete for unit disk graphs. A proof of the NP-Completeness of the MLAS problem for unit disk graphs was already given earlier in [8], but our proof is simpler and easier to follow because of the fact that we give a sequence of two reductions.

Chapter 3

Optimal and Approximation Algorithms for Specific Topologies

In this chapter, we start by proving that the MLAS problem is NP-Complete, even for unit disk graphs. We then study specific topologies for which we can either find optimal algorithms or approximation algorithms with proven approximation ratios.

A trivial lower bound for any topology is the maximum distance from any node to the sink. This measure is sometimes referred to as the network radius R [37, 21]. Since at most half of the nodes that have not yet transmitted can transmit at any time step, $\log n$ is also a lower bound on the latency, as observed in [37]. For specific topologies, we will show that we can improve on the above lower bounds and give new lower bounds for trees, grids, tori and unit interval graphs.

3.1 NP-Completeness Proof

In this section, we study the decision version of the MLAS problem which we call the Aggregation Convergecast Problem: given an integer k and a graph $G = (V, E)$ of size n where $V = \{S_0, S_1, \ldots, S_{n-1}\}$ and where S_{n-1} is the sink node, is there a valid

schedule that aggregates the data from all the nodes to the sink in k time slots.

Theorem 2 *The Aggregation Convergecast Problem is NP-Complete.*

Proof. We prove it by reducing from the 3-SAT problem.

Consider a boolean formula f in 3-CNF, where $\{x_1, x_2, \ldots, x_n\}$ is the set of boolean variables and $\{c_1, c_2, \ldots, c_m\}$ is the set of clauses. We create an instance of the aggregation convergecast problem by creating a graph G_f as follows :

1. For each variable x_i, we create a *line* X_i with the set of nodes $\{x_{i,1}, x_{i,2}, \ldots, x_{i,m}, x_i, \bar{x}_{i,m}, \bar{x}_{i,m-1}, \ldots, \bar{x}_{i,1}\}$.

2. For every node $x_{i,j}$ (resp. $\bar{x}_{i,j}$) created in 1, we create a node $a_{i,j}$ (resp. $\bar{a}_{i,j}$) and a path $P_{i,j}$ (resp. $\bar{P}_{i,j}$) of length $l_{i,j} = max(i-1, 1) + max(j-2, 0)$ between $a_{i,j}$ and $x_{i,j}$ (resp. $\bar{a}_{i,j}$ and $\bar{x}_{i,j}$). For further reference, we denote by $b_{i,j}$ (resp. $\bar{b}_{i,j}$) the node on the path $P_{i,j}$ (resp. $\bar{P}_{i,j}$) that is connected to $x_{i,j}$ (resp. $\bar{x}_{i,j}$), and we denote by gadget g_i the set of nodes contained in X_i and all the paths $P_{i,j}$.

3. Each gadget g_i is connected to g_{i+1} by connecting x_i to x_{i+1} for $1 \leq i < n$.

4. For each clause c_j, we create a node c_j.

5. For every literal x_i (resp. $\bar{x}_i$) that appears in clause c_j, we connect node c_j to $a_{i,j}$ (resp. $\bar{a}_{i,j}$).

6. The sink node is node x_n.

This transformation can be done in polynomial time as the number of nodes created in G_f is less than $2m^2 + 2mn + m + n$.

We now show that f is satisfiable if and only if there exists a tree and schedule that completes the aggregation convergecast of G_f in $m + n + 1$ time slots.

Suppose that f has a satisfying assignment A. For each clause $c_j = (l_{j1} \vee l_{j2} \vee l_{j3})$ in f, we know that at least one of l_{j1}, l_{j2}, l_{j3} must be true in A. Without loss of generality, assume that l_{j1} is true, and that it represents the unnegated literal x_i (resp. negated literal $\bar{x}_i$). The convergecast tree is built by keeping the link between node c_j and node $a_{i,j}$ (resp. $\bar{a}_{i,j}$), and by removing the other edges connected to c_j. If more than one literal in clause c_j is true, then node c_j connects to only one of the corresponding $a_{i,j}$ (resp. $\bar{a}_{i,j}$) nodes. Clearly, all c_j nodes are leaves in the tree, and the only other leaves are a subset of the $a_{i,j}$ and $\bar{a}_{i,j}$ nodes. The schedule is built as follows:

1. Node c_j is scheduled to transmit at time 1 for $1 \leq j \leq m$.

2. Node $a_{i,j}$ (resp. $\bar{a}_{i,j}$) is scheduled to transmit at time 1 if x_i (resp. $\bar{x}_i$) is false in A, and at time 2 if x_i (resp. $\bar{x}_i$) is true in A.

3. Intermediate nodes on the tree are scheduled to transmit one time slot after they've received from all their children.

This tree and schedule is illustrated in Figure 3 and a specific instance of the problem is shown in Figure 4.

We now prove that such a schedule is collision-free and that it completes the aggregation convergecast in $m + n + 1$ time slots.

Since the collisions always happen at the receiver's end, we only need to consider the nodes that have more than one child in the convergecast tree. First, we show that there is no collision at nodes $x_{i,j}$ (resp. $\bar{x}_{i,j}$) for $1 \leq i \leq n$ and $1 < j \leq m$, i.e. we show that $x_{i,j-1}$ (resp. $\bar{x}_{i,j-1}$) does not transmit at the same time as $b_{i,j}$ (resp. $\bar{b}_{i,j}$). Consider the case where x_i is false in A, $x_{i,1}$ will receive from its only child at time $l_{i,1} = max(i-1, 1) + max(1-2, 0) = max(i-1, 1)$, and will transmit to node $x_{i,2}$ at time $max(i-1, 1) + 1$. Node $x_{i,2}$ receives from $b_{i,2}$ at time

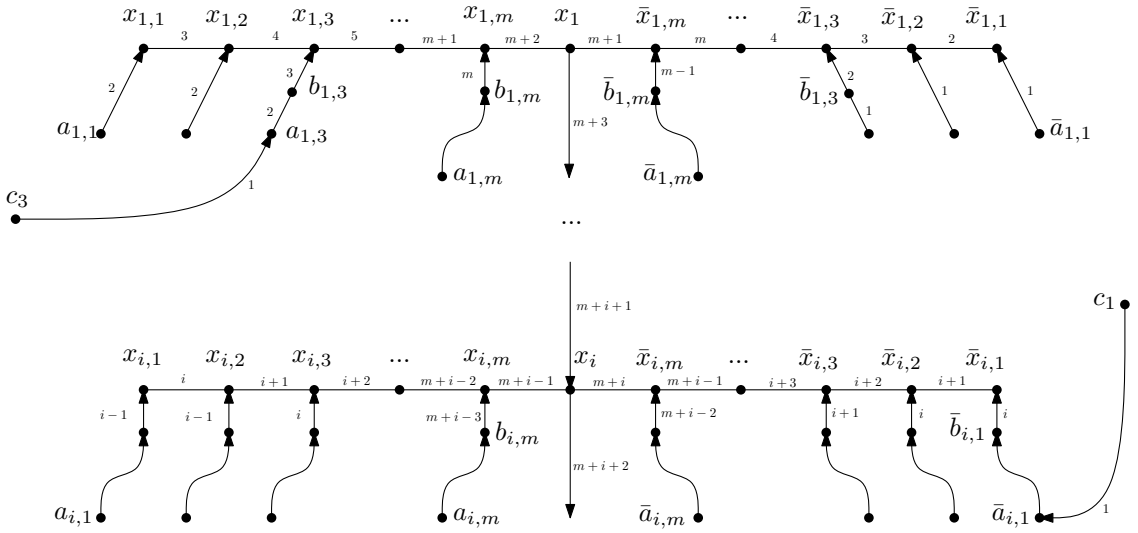

Figure 3: Illustration of how the schedule is built for a generic 3-SAT problem where $f = (\bar{x}_i \vee \ldots) \wedge (\ldots) \wedge (x_1 \vee \ldots) \wedge \ldots$ and $A = \{x_1 = T, \ldots, x_i = F, \ldots\}$.

$l_{i,2} = max(i-1,1) + max(2-2,0) = max(i-1,1)$. Clearly, there is no collision at node $x_{i,2}$ as its children use different time slots. By induction, we can show that there is no collision at $x_{i,j}$ for $2 \leq j \leq m$. Indeed, we can easily see that $x_{i,j-1}$ will transmit to $x_{i,j}$ at time $max(i-1,1) + j - 1$, and that $b_{i,j}$ will transmit at time $l_{i,j} = max(i-1,1) + j - 2$.

When x_i is true in A, we can show that there is no collision by using a similar demonstration. The nodes will simply transmit one time slot later (when compared to the case where x_i is false), because the $a_{i,j}$ nodes transmit at time 2 instead of time 1. This ensures that node $x_{i,m}$ will not transmit at the same time as node $\bar{x}_{i,m}$, because one will transmit at time $max(i-1,1) + m$ and the other one will transmit at time $max(i-1,1) + m + 1$. For node x_1, that means it receives from one of its children at time $m+1$, from the other child at time $m+2$, and that it is scheduled to transmit to x_2 at time $m+3$.

Next we show that there is no collision at x_i for $2 \leq i \leq n$. Assume inductively that node x_{i-1} transmits to node x_i at time $m+i+1$. The other 2 children of x_i are

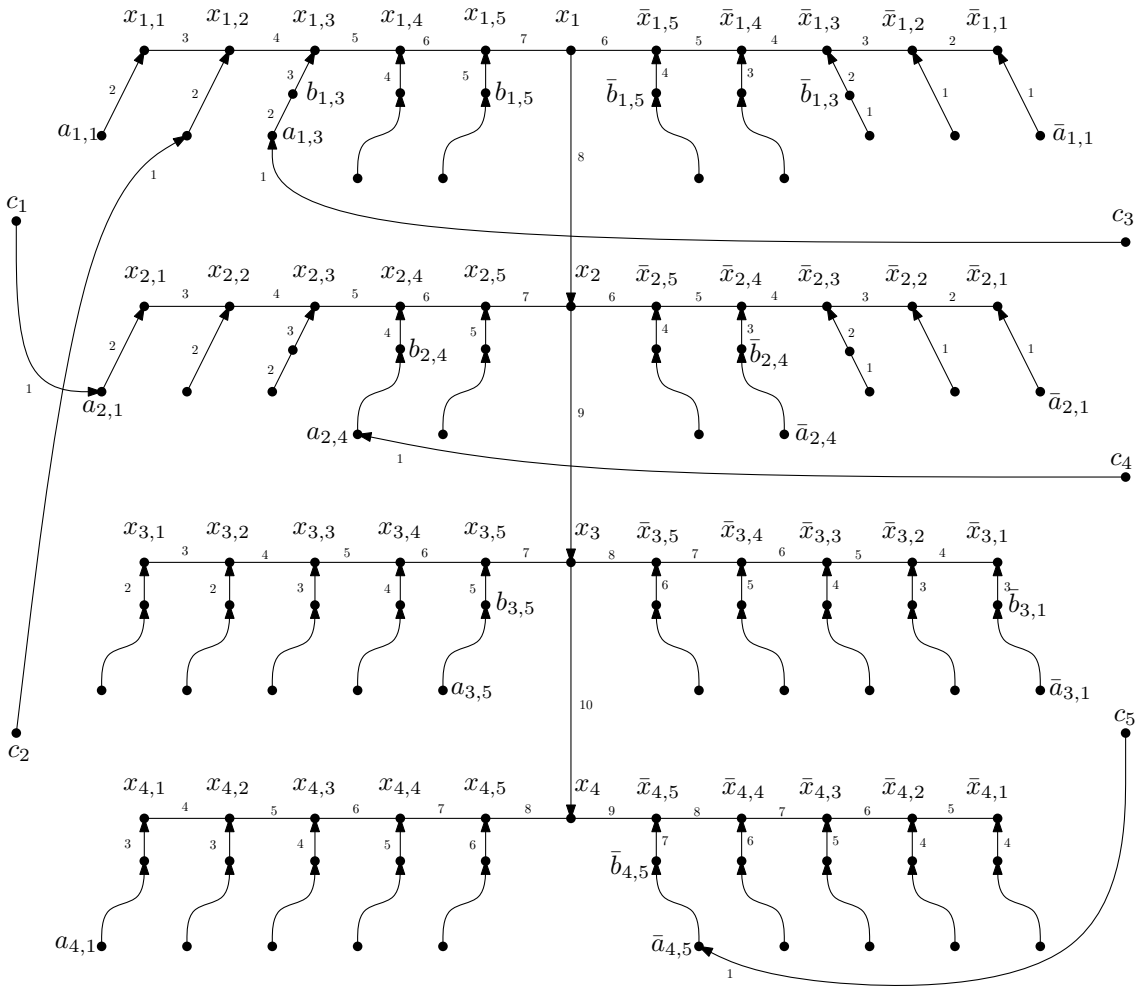

Figure 4: Example tree and schedule built for a specific instance of 3-SAT where $f = (\bar{x}_1 \vee x_2 \vee x_4) \wedge (x_1 \vee \bar{x}_3 \vee x_4) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_4) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_3 \vee \bar{x}_4)$ and $A = \{x_1 = T, x_2 = T, x_3 = F, x_4 = F\}$. Clearly, there are no collisions and the aggregation convergecast is completed in $m + n + 1 = 5 + 4 + 1 = 10$ time slots.

transmitting at time $max(i-1,1)+m = m+i-1$ and time $max(i-1,1)+m+1 = m+i$. It is straightforward to see that there is no collision at x_i and that the last node to transmit will be node x_{n-1} and that it will transmit at time $m+n+1$.

This proves that the schedule is collision-free and that it completes the aggregation convergecast in $m+n+1$ time slots.

Now suppose that G_f has a valid schedule that completes the aggregation convergecast in $m+n+1$ time slots using the tree T. For each node $c_j \in G_f$, we take its parent in T and use it to determine which variables are true in f. Let $n_{i,j} \in P_{i,j}$ (resp. $\bar{n}_{i,j} \in \bar{P}_{i,j}$) be the parent of c_j in T, we assign true (resp. false) to x_i in f. Since every node c_j must have a parent in T, we know that every clause in f will have at least one literal that makes it true. We need to show that this is a valid assignment, i.e. there is no situation in which x_i would be set to true and false.

The gadgets g_i are built to generate a collision at x_i whenever the unnegated and the negated sides of the gadget are connected to a node c_j in T. Suppose instead that there is a solution in which c_j is connected to $n_{i,j} \in P_{i,j}$ and c_k is connected to $\bar{n}_{i,k} \in \bar{P}_{i,k}$. We've shown earlier that when the first node in a path $P_{i,j}$ transmits at time 2, the earliest its information can reach node x_i is at time $max(i-1,1)+m+1$. Since both $n_{i,j}$ and $\bar{n}_{i,k}$ have to aggregate the information from their respective child, their aggregated information could reach node x_i at the same time and one of $x_{i,m}$ or $\bar{x}_{i,m}$ will have to wait at least one time slot to transmit to x_i. If $i = 1$, this means either $x_{1,m}$ or $\bar{x}_{1,m}$ transmits at time $m+3$. Therefore, x_1 cannot transmit before time $m+4$ and its information cannot reach the sink before $m+4+dist(x_1, x_n) = m+n+2$. If $i > 1$, either $x_{i,m}$ or $\bar{x}_{i,m}$ cannot transmit before time $m+i+1$. However, this is precisely the earliest time that x_{i-1} also can transmit, so one of them will have to transmit at time $m+i+2$ to avoid a collision at x_i. Thus, x_i can only transmit at time $m+i+3$ at the earliest, which implies that its information won't reach the sink

before time $m + i + 3 + dist(x_i, x_n) = m + n + 2$. This contradicts the assumption that the schedule completes the aggregation convergecast in $m + n + 1$ time slots, and proves that there is no situation in which a variable x_i would be set to true and false.

■

We now show that the aggregation convergecast problem is also NP-Complete for unit disk graphs by doing some changes to our proof. Instead of reducing from the general 3-SAT problem, we reduce from a restricted version of the planar 3-SAT problem, introduced by Lichtenstein [28]. This restricted version of the problem has the following additional properties:

1. Each variable occurs in at most three clauses.
2. Both unnegated and negated instances of each variable appear.
3. For every variable node x in G_f, all the edges representing positive instances of the variable are incident to one side of x and all the edges representing negative instances are incident to the other side of x.

We first transform the planar formula graph G_f by replacing each variable node x_i by a gadget g_i, and adjusting the edges using the same logic as in the previous proof. Since G_f is planar and because of the constraints of the restricted planar 3-SAT problem, it is relatively easy to see that we can draw the transformed graph in such a way that there is no crossing of edges. It is also easy to see that the maximum degree of the transformed graph is 4.

We then use the following lemma from Chen et al.[8] to transform the graph into a unit disk graph:

Lemma 1 *Let H be a plane graph on g vertices with a maximum degree at most 4. Suppose that H is not an octahedron, and let H' be the graph obtained from H by*

replacing each edge in H with a path of length $120g^2$. Then H' is a unit disk graph and an orthogonal planar embedding of H' of grid size $(40g^2 + 40g) \times (40g^2 + 40g)$ can be computed in time polynomial in g.

Since H' is built by replacing each edge by a path of constant length $120g^2$, we can use the same argument we used for the general aggregation convergecast problem to prove that the restricted 3-SAT problem is satisfiable if and only if we can build a spanning tree T of the unit disk graph H' with a schedule that completes the aggregation convergecast in $(120g^2)(m + n + 1)$ time slots. This yields the following theorem:

Theorem 3 *The Aggregation Convergecast Problem is NP-Complete even for unit disk graphs.*

3.2 Trees

In this section, we show that finding lower bounds for specific tree topologies is relatively straightforward and we give an algorithm for building an optimal schedule for the MLAS problem.

In tree networks, building a conflict-free schedule is simplified by the fact that conflicts can only occur between the children of a node. For example, consider a binary tree in which all leaf nodes are at the same depth and all internal nodes have two children. Such a tree is called a *perfect binary tree* and it is not hard to show using induction on the number of levels in the tree that $2\lfloor \log_2 n \rfloor$ is a lower bound on the latency of the schedule. In other words, in a perfect binary tree, two time slots are required per level. An example of an optimal schedule for a perfect binary tree is shown in Figure 5. We can generalize this observation to perfect k-ary trees and

say that an optimal schedule for a given perfect k-ary tree would have a latency of $k\lfloor \log_k n \rfloor$ (see Figure 6).

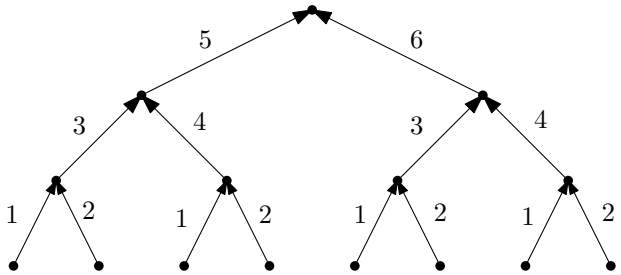

Figure 5: Example of an optimal schedule for a perfect binary tree. The schedule uses exactly $2\lfloor \log_2 n \rfloor = 6$ time slots.

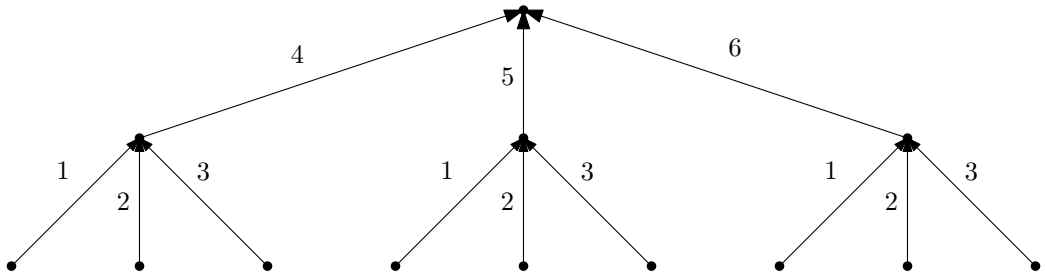

Figure 6: Example of an optimal schedule for a perfect ternary tree. The schedule uses exactly $3\lfloor \log_3 n \rfloor = 6$ time slots.

Because of the fact that conflicts can only happen between children of a node, the MLAS problem for trees is similar to the broadcast problem in wired networks. For this reason, the broadcast algorithm proposed in [34] could be used to find an optimal schedule for the MLAS problem on a given tree. Still, we present our own algorithm here as it was developed independently and for a different model.

We now introduce our ScheduleTree algorithm (See Algorithm 2), which builds optimal schedules for arbitrary trees.

Theorem 4 *Given a tree T rooted at the sink, the* ScheduleTree *algorithm (see Algorithm 2) builds a conflict-free schedule that solves the MLAS problem optimally. This schedule can be built in $\mathcal{O}(|V| \log |V|)$ time.*

Algorithm 2 ScheduleTree

Input: v
Output: $v.t$

```
1: if |C(v)| == 0 then
2:     v.t = 1
3: else
4:     for each c ∈ C(v) do
5:         ScheduleTree(c)
6:     end for
7:     Sort(C(v))                    ▷ sort children in order of assigned time slot
8:     i = 2
9:     while i < |C(v)| do
10:        u = C_i(v)
11:        x = C_{i-1}(v)
12:        u.t = Max(x.t + 1, u.t)
13:        i = i + 1
14:    end while
15:    u = C_{i-1}(v)
16:    v.t = u.t + 1
17: end if
```

Proof. Algorithm 2 is called recursively until it reaches a leaf node, at which point it will assign time slot 1 to the leaf. For any node $v \in T$, assume inductively that v's children are assigned a time slot which corresponds to the time at which v has aggregated all the information from its children and is ready to transmit. At this point, some children might have been assigned the same time slot. To resolve those potential conflicts, we first sort the children of v in order of currently assigned time slot (line 7). The loop of lines 9-14 will then resolve conflicts by assigning a different time slot to each child. At each iteration starting with the second child, the child either keeps its currently assigned time slot or it is scheduled to transmit one time slot after the previous child, depending on which is greater. This guarantees that each child will transmit at a different time slot, which means the children's schedule is now conflict-free. The loop ensures that each child is assigned the first available time slot after it is ready to transmit. The children's schedule is thus optimal. At

the end of this for loop, the children are still sorted in order of assigned time slot. Line 16 assigns to v the time slot that follows the one used by its last child. This ensures that v transmits as soon as it has received all the data from its children, and completes the proof by induction.

We use amortized analysis to count the total number of operations. At every node, the amount of time needed is $\mathcal{O}(i \log i)$ where i is the size of the adjacency list of the node. The total amount of time is therefore $\mathcal{O}(|E| \log |E|)$ where $|E|$ is the total number of edges. This in turn is $\mathcal{O}(|V| \log |V|)$ since the graph is a tree. ■

3.3 Grids and Tori

Depending on the position of the sink and the size of the grid, there are many cases in which a schedule can be built by using exactly R time slots. However, there are cases where this is not possible because of conflicts when building the schedule. In a grid, conflicts will occur if the sink is exactly in the middle of a row or a column. When that happens, at least one node will have to be scheduled one time slot after it is ready to transmit to avoid a conflict with another node. This leads us to our lower bound for grids.

Theorem 5 *Given a grid G of 2 dimensions, any MLAS scheduling algorithm requires a minimum number of time slots defined by $T_{min} = max\{dist(S_i, s) : i = 1, 2, \ldots, n\} + C_{col} + C_{row}$, where C_{col} equals 1 if the sink is exactly in the middle of a row and 0 otherwise, and where C_{row} equals 1 if the sink if exactly in the middle of a column, and 0 otherwise.*

Proof. Consider a grid G of m rows by n columns. We have three cases to look at in order to show that the lower bound holds. We need to look at the case where the sink is located in the middle row, the case where the sink is located in the middle

column, and the case where the sink is located in the middle row and column at the same time.

Let m be odd and let the position of the sink be $(\lceil \frac{m}{2} \rceil, j)$ where j is not the middle column. There are 2 nodes that are located at a distance of R time slots from the sink. If $j < \lceil \frac{n}{2} \rceil$, those 2 nodes are located at $(1, n)$ and (m, n), and if $j < \lceil \frac{n}{2} \rceil$ then those 2 nodes are located at $(1, 1)$ and $(m, 1)$. If these two nodes' data follow a disjoint path, then the last nodes on each path will be ready to transmit at the same time and one of them will have to be scheduled at least one time slot later. Suppose instead that their paths merge at some point. No matter where those paths merge, if they are shortest paths, we still have a conflict to resolve at the merge point and a delay of at least one time slot is inevitable. If one of the path is not a shortest path, then this path has a minimum length of $R + 1$ and our lower bound still holds. The same argument also holds when the sink is located at $(i, \lceil \frac{n}{2} \rceil)$ where i is not the middle row.

This leads us to the last case where the sink is exactly in the middle, in which case the 4 corners are R time slots away from the sink. If their data follow disjoint paths, 4 nodes will be ready to transmit to the sink at the same time and the last scheduled node will transmit at $R+3$. However, we can do better by making the paths converge before the sink. We still have conflicts at the merge points, but those conflicts can be handled independently if we merge 2 paths at one point and the other 2 paths at another point. Therefore, we only need one time slot to handle both conflicts and the merged paths now have a schedule length of $R + 1$. We have one more conflict to resolve at the sink, but since we now only have 2 paths, we only need one more time slot which gives us the lower bound of $R + 2$. ■

We can use the same logic to obtain the following lower bound, generalized for grids of k dimensions.

Corollary 6 *Given a grid G of k dimensions, any MLAS scheduling algorithm requires a minimum number of time slots defined by $T_{min} = max\{dist(v_i, s) : i = 1, 2, \ldots, n\} + \sum_{j=1}^{k} C_j$, where C_j equals 1 or 0 depending on the position of the sink in the grid.*

$$C_j = \begin{cases} 1 & \text{if } G.size_j > 1 \text{ and } sink.pos_j == (G.size_j + 1)/2 \\ 0 & \text{otherwise} \end{cases}$$

The same argument also holds for torus networks, although the sink position is not important for this topology. Because of the circular structure of a torus, the sink is always exactly in the middle of a dimension when the size of the dimension is odd. If the size of the dimension is even, the sink cannot be exactly in the middle so there is no possible conflict. This leads us to the following corollary.

Corollary 7 *Given a torus G of k dimensions, any MLAS scheduling algorithm requires a minimum number of time slots defined by $T_{min} = \sum_{i=1}^{k} \lfloor \frac{G.size_i}{2} \rfloor + \sum_{j=1}^{k} C_j$, where C_j equals 1 or 0 depending on the size of each dimension:*

$$C_j = \begin{cases} 1 & \text{if } G.size_j > 1 \text{ and } G.size_j \text{ is odd} \\ 0 & \text{otherwise} \end{cases}$$

We now present our algorithm for solving the MLAS problem for grid topologies. The algorithm starts by building a Shortest Path Tree T rooted at the sink. For a 2-dimensional grid, T is built by keeping the vertical edges that are in the same column as the sink, as well as all the horizontal edges. The scheduling logic is relatively simple as conflicts can only happen in nodes located in the same column as the sink. Indeed, those are the only nodes in T with more than one children. If the sink is not located in the middle of a dimension, we don't need to bother about conflicts and we can simply assign the first valid time slot for every node. If the sink is in the middle column, then we need to handle conflicts at all the nodes in the column of the sink. If the sink is in the middle row, we need to handle a conflict at the sink node. Figure 7

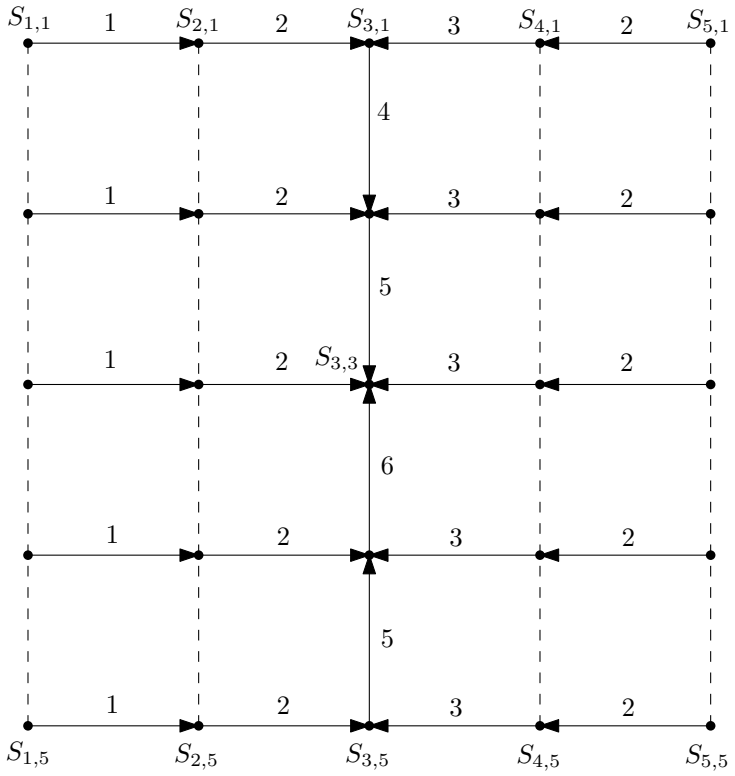

Figure 7: Example of a schedule built by the ScheduleGrid algorithm for a 5×5 grid with the sink node located at $(3, 3)$. As you can see, nodes at $(2, j)$ and $(4, j)$ need to be assigned a different time slot to avoid a collision at $(3, j)$. The nodes at $(3, 2)$ and $(3, 4)$ also have to transmit at different time slots to avoid a collision at the sink.

illustrates a case where the sink is in the middle of both dimensions and shows how our algorithm handles the conflicts.

The ScheduleGrid algorithm works as follows. Line 2 checks if there is an equal number of nodes to the left and to the right of the sink. If this is the case, that means that there is a potential for conflict and that we need to add 1 to the time slots assigned to one side of the sink (the right side in our algorithm). Line 3 assigns 1 to C_{col} in that case, and C_{col} will be later added to the time slots assigned to the nodes on the right side of the sink. Line 5 checks if there is an equal number of nodes below and above the sink. If true, line 6 assigns 1 to C_{row} which will be later added to the time slots assigned to the nodes below the sink.

Algorithm 3 ScheduleGrid

Input: S: 2-dimensional array of nodes, s: sink node
Output: Assignment of parent and time slot for each node in S

1: $C_{col} = C_{row} = 0$
2: **if** $s.x - 1 == G.sizeX - s.x$ **then**
3: $\quad C_{col} = 1$
4: **end if**
5: **if** $s.y - 1 == G.sizeY - s.y$ **then**
6: $\quad C_{row} = 1$
7: **end if**
8: $maxtx = Max(s.x - 1, G.sizeX - s.x) + C_{col}$
9: **for** $y = 1$ to $G.sizeY$ **do**
10: $\quad$ **for** $x = 1$ to $G.sizeX$ **do**
11: $\quad\quad$ **if** $x < s.x$ **then**
12: $\quad\quad\quad S[x, y].p = S[x + 1, y]$
13: $\quad\quad\quad S[x, y].t = x$
14: $\quad\quad$ **else if** $x > s.x$ **then**
15: $\quad\quad\quad S[x, y].p = S[x - 1, y]$
16: $\quad\quad\quad S[x, y].t = G.sizeX - s.x - x + C_{col}$
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: $\quad$ **if** $y < s.y$ **then**
20: $\quad\quad S[s.x, y].p = S[s.x, y + 1]$
21: $\quad\quad S[s.x, y].t = maxtx + y$
22: $\quad$ **else if** $x > s.x$ **then**
23: $\quad\quad S[s.x, y].p = S[s.x, y - 1]$
24: $\quad\quad S[s.x, y].t = maxtx + G.sizeY - s.y - y + C_{row}$
25: $\quad$ **end if**
26: **end for**

Line 8 computes the maximum time slot used by any horizontal edge. This value is used when scheduling nodes in the column of the sink since they need to be scheduled after the data form their children has been aggregated. Lines 9-26 build the tree and the schedule for the graph. The inner loop of lines 10-18 will create and schedule the horizontal edges. As you can see at line 16, C_{col} is added to the time slot assigned to each node that is located at the right of the sink ($x > sink.x$) to handle potential conflicts. Lines 19-25 create and schedule the vertical edges. The $maxtx$ variable computed earlier is used at lines 21 and 24 to ensure that those nodes are scheduled after they have received from their children. The C_{row} variable is added to each node that is located below the sink ($y > sink.y$) to handle potential conflicts.

Since the algorithm iterates through all the nodes only once and the processing done at each node is constant, the ScheduleGrid algorithm has a time complexity of $\mathcal{O}(|V|)$.

We can clearly see that the schedule built by the ScheduleGrid algorithm is conflict-free and uses a number of time slots equal to our lower bound defined in Theorem 5. The schedule is thus conflict-free and optimal. As we've shown earlier, a torus network can be treated as a grid network where the sink is in the middle. Therefore, the same algorithm can be used to build a schedule for tori. This leads to the following theorem:

Theorem 8 *Given a grid or torus network of k dimensions, the* ScheduleGrid *algorithm (described in Algorithm 3) builds a conflict-free schedule that solves the MLAS problem optimally. This schedule can be built in* $\mathcal{O}(|V|)$ *time.*

3.4 Unit Interval Graphs

Sometimes sensor nodes can be deployed to monitor perimeters or borders and their positions therefore fall on a line. Two nodes are connected if each is contained in the other's transmission range. If all transmission ranges are equal, they can be represented by unit intervals and the resulting graph can be represented by a unit interval graph[1].

We consider a unit interval graph $G = (V, E)$ of size n, where $V = \{S_0, ..., S_{n-1}\}$ and where S_{n-1} is the sink node. We assume that all sensor nodes are located at distinct locations. The nodes are sorted in descending order of distance from the sink which means S_0 is the farthest node from the sink. S_j is called a forward neighbor of S_i if it is closer to the sink than S_i (i.e. if $j > i$), otherwise it is called a backward neighbor.

We also study a more constrained kind of unit interval graph where all nodes except for the last k nodes have k forward neighbors. We call such a graph a *regular unit interval graph.*

In this section, we look at unit interval graphs and regular unit interval graphs, and we give lower bounds and algorithms for both topologies.

3.4.1 Lower Bound for Unit Interval Graphs

In a unit interval graph, cliques have special properties that allow us to find a tighter lower bound.

Lemma 2 *In a clique, if a node S'_i is receiving from a node S_i, no other node inside the clique can transmit at the same time.*

[1] In the usual definition of an interval graph, intervals are represented by nodes and there is an edge between two nodes if their corresponding intervals overlap. It is easy to see that our definition is equivalent.

Proof. Any node inside the clique that would transmit at the same time would interfere at S'_i. ∎

Lemma 3 *In a unit interval graph, 2 nodes S_i and S_j with $i < j$ that are part of the same clique can transmit at the same time if and only if S_i transmits backwards to a node S'_i that is outside the clique and S_j transmits forwards to a node S'_j outside the clique.*

Proof. If $i' > i$, then S_j's transmission will interfere with S'_i's reception. If $j' < j$, then S_i's transmission will interfere with S'_j's reception. This proves that S_i has to transmit backwards and that S_j has to transmit forwards. And we know from Lemma 2 that neither S'_i nor S'_j can form a clique with S_i and S_j. ∎

Obviously, the data sent by a node has a minimum latency that is equal to the minimum number of hops to the sink. Given nodes $S_0, \ldots, S_{n-1}$ in a unit interval graph, we denote by $dist(S_i, S_{n-1})$ the minimum number of hops between a node S_i and the sink.

However, the hop distance alone is not the only factor to consider if we want to have a good lower bound. Indeed, a node cannot transmit before it has received all the data from its children, and it needs to avoid collisions with other transmissions. Suppose a node S_i transmits at time t, then its data cannot reach the sink before:

$$dist(S_i, S_{n-1}) - 1 + t$$

We denote by $ln(S_i)$ the last neighbor of S_i, i.e. the neighbor of S_i that is the closest to the sink. Consider the nodes to be divided into cliques $C_0, ..., C_{m-1}$ where $C_0 = \{S_0, ..., ln(S_0)\}$, $C_1 = \{ln(S_0), ..., ln(ln(S_0))\}$ and so on. Note that, as defined, the cliques are overlapping (not disjoint); the last node in clique C_i is the first node

in clique C_{i+1}, for $0 \leq i < m-1$. Figure 8 shows an example of a graph with three cliques.

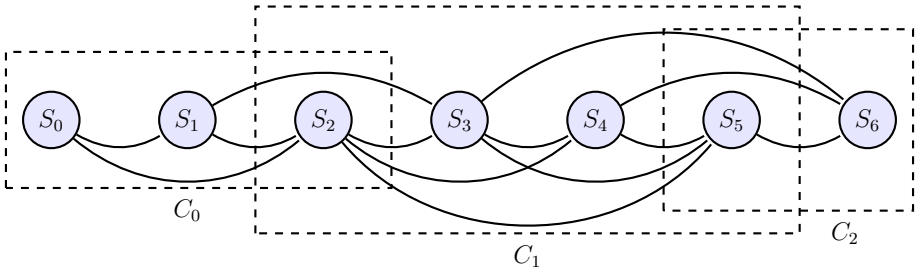

Figure 8: Illustration of a graph with 3 cliques where $C_0 = \{S_0, S_1, S_2\}$, $C_1 = \{S_2, S_3, S_4, S_5\}$, $C_2 = \{S_5, S_6\}$.

Based on these definitions, we can now introduce the following lower bound:

Theorem 9 *In a unit interval graph of size n, any MLAS scheduling algorithm must use a minimum number of time slots defined by the following formula :*

$$max \begin{cases} |C_0| + dist(C_0, S_{n-1}) - 1 \\ |C_{m-1}| - 1 \\ \frac{|C_i|+|C_{i+1}|}{2} + dist(C_i, S_{n-1}) - 1 \end{cases}$$

Proof. We know from Lemma 3 that no more than 2 nodes in a clique can transmit at the same time. Because C_0 doesn't have a clique behind it, no node can transmit backwards outside the clique and so only one node in C_0 can be scheduled at a given time slot. Similarly, only one node in C_{m-1} can be scheduled at a given time slot. It follows from the fact that no node in C_{m-1} has a forward neighbor outside of the clique.

Thus, no matter how we schedule the nodes, there is no way to use less time slots than the size of C_0 or the size of C_{m-1} minus 1 (because the sink is part of C_{m-1}). Suppose that $S_i \in C_0$ is the last node in that clique to transmit, the earliest it can transmit is at time $|C_0|$, and the earliest its data can reach the sink is at time

$|C_0|+dist(C_0, S_{n-1})-1$. Similarly, the last node to transmit in C_{m-1} cannot transmit before time $|C_{m-1}|-1$ and this is also the earliest its data can reach the sink because $dist(C_{m-1}, S_{n-1}) = 0$.

This takes care of the first 2 lower bounds in our formula. The third lower bound is a little trickier to prove as C_1 through C_{m-2} have the possibility to schedule 2 nodes to transmit at the same time slot. This would give us a straightforward lower bound of $\frac{|C_i|}{2}+dist(C_i, S_{n-1})$. However, this trivial bound is overoptimistic because it doesn't take the neighboring cliques into consideration. Indeed, any time slot used to transmit to a neighboring clique prevents any node in the receiving clique to transmit at the same time. Therefore, it effectively adds one to the number of time slots used by this neighboring clique.

Consider 2 consecutive cliques C_i and C_{i+1}. From Lemma 3, we know that a maximum of 2 nodes in a clique can transmit at the same time. Suppose that 2 nodes in C_i transmit at time t. One of these nodes has to transmit to a node in C_{i+i}. Therefore, we know from Lemma 2 that no node in C_{i+1} can transmit at time t.

Similarly, suppose that 2 nodes in C_{i+1} transmit at time t'. One of these nodes has to transmit to a node in C_i. Therefore, no node in C_i can transmit at time t'. This shows that no more than 2 nodes in $C_i \cup C_{i+1}$ can transmit at the same time.

It is now easy to see that we need a minimum of $\frac{|C_i|+|C_{i+1}|}{2}$ time slots for each pair of consecutive cliques. To tighten this bound, we need to add the distance between the pair of cliques and the sink minus 1, which gives us exactly the third lower bound in our formula.

■

3.4.2 Algorithm for Unit Interval Graphs

In this section, we present our 2-approximation algorithm for solving the MLAS problem in Unit Interval Graphs.

A simple greedy approach for building the convergecast tree would be for each node to select as parent the neighbor that is the closest to the sink node. This approach leads to bad results in practice because it reduces the chance of being able to schedule many nodes to transmit at the same time slot.

An approach that was found to give good results in practice was to divide the nodes in cliques and to select one node in each clique as the aggregator for the data from all the other nodes of the same clique. This is the approach we use in our Hub Algorithm and it is illustrated in Figure 9.

The Hub Algorithm works by dividing the graph into cliques $C_0, ..., C_{m-1}$ where $C_0 = \{S_0, ..., ln(S_0)\}$, $C_1 = \{ln(S_0), ..., ln(ln(S_0))\}$ and so on. As was the case in our lower bound definition, the cliques are overlapping in such a way that the last node in clique C_i is the first node in clique C_{i+1}, for $0 \leq i < m-1$. The last node in each clique is used as the aggregator for the clique. The schedule is built by iterating through all the nodes and assigning to each node the lowest time slot not already in use by a node in the previous and current clique.

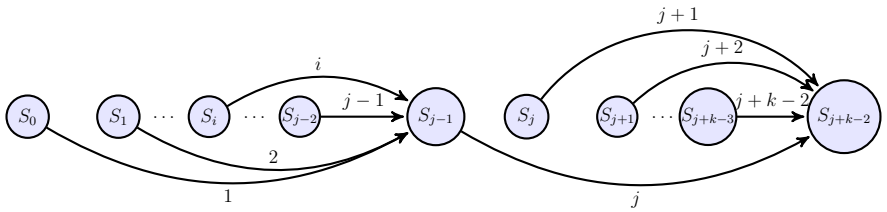

Figure 9: In the Hub algorithm, intermediate nodes are selected and used as aggregators along the way to the sink. In this figure, $j = |C_0|$ and $k = |C_1|$. Note that the total number of time slots used by these 2 cliques is $j + k - 1$, assuming that S_{j+k-1} is not the sink and is scheduled at time $j + k - 1$.

Theorem 10 *The Hub Algorithm is a 2-approximation algorithm and builds a schedule in $\mathcal{O}(|V|)$ time.*

Proof. Using the algorithm definition, we can easily determine the number of time slots used by any pair of 2 consecutive cliques. Let C_i and C_{i+1} be 2 consecutive cliques, the number of time slots used by their nodes is $|C_i| + |C_{i+1}| - 1$, or the total number of nodes in those 2 cliques. Thus, the total latency of the algorithm is given by $Max(|C_i| + |C_{i+1}| - 1 + dist(C_i, S_{n-1}))$.

In the worst case, this is no more than twice the number of time slots defined by our lower bound of $\frac{|C_i|+|C_{i+1}|}{2} + dist(C_i, S_{n-1}) - 1$. This proves that the Hub Algorithm is a 2-approximation algorithm.

Assuming that the nodes are already sorted in order of index, the cliques can be found in $\mathcal{O}(|V|)$ time in the worst case. Every node is assigned as parent the last node in its clique which can be done in constant time. By maintaining separate lists for the set of time slots used in the previous two cliques and a set of available time slots, the schedule can be built in $\mathcal{O}(|V|)$ time. ■

3.4.3 Lower Bound for Regular Unit Interval Graphs

In this section, we show that we can find a tighter lower bound for the regular unit interval graph. We start by observing that in a regular unit interval graph, each set of $k + 1$ consecutive nodes represents a clique. This follows from the fact that each node has a maximum of k forward neighbors and k backward neighbors.

Obviously, the data sent by a node has a latency that is greater or equal to the number of hops to the sink. Given nodes $S_0, \ldots, S_{n-1}$ in a regular unit interval graph, the number of hops between a node and the sink can be found with the following

formula:

$$dist(S_i, S_{n-1}) = \left\lceil \frac{n-1-i}{k} \right\rceil$$

But the hop distance alone is not the only factor to consider if we want to have a good lower bound. Indeed, a node cannot transmit before it has received all the data to aggregate and it needs to avoid collisions with other transmissions. Suppose a node S_i transmits at time t, then its data cannot reach the sink before:

$$dist(S_i, S_{n-1}) - 1 + t$$

Based on these definitions, we can now introduce the following straightforward lower bound:

Theorem 11 *In a regular unit interval graph of size n where each node has a maximum of k forward neighbors, any aggregation convergecast algorithm must use a minimum of* $\left\lceil \frac{n-1}{k} \right\rceil + k - 1$ *time slots.*

Proof. The first $k+1$ nodes form a clique, and it follows from Lemma 3 that only one node of this group can transmit at a time. Therefore, $k+1$ time slots are needed for these nodes to transmit their data. Suppose that $S_i \in \{S_0, \ldots, S_k\}$ is the last node in that group to transmit, the earliest it can transmit is at time $k+1$, and the earliest it can reach the sink is at time

$$\begin{aligned} dist(S_i, S_{n-1}) - 1 + k + 1 &= \left\lceil \frac{n-1-i}{k} \right\rceil + k \\ &\geq \left\lceil \frac{n-1-k}{k} \right\rceil + k \\ &= \left\lceil \frac{n-1}{k} \right\rceil + k - 1 \end{aligned}$$

■

We now show that the above lower bound can be strengthened for large enough values of n and k. Let $n \geq 2k+3$, $k > 2$ and consider the first $3k+3$ nodes to be divided into 3 cliques of size $k+1$, where $C_0 = \{S_0, \ldots, S_k\}$, $C_1 = \{S_{k+1}, \ldots, S_{2k+1}\}$ and $C_2 = \{S_{2k+2}, \ldots, S_{3k+2}\}$.

Theorem 12 *In a regular unit interval graph of size n where each node has a maximum of k forward neighbors, if* $n \geq 2k+3$, $k > 2$ *and* $(n-1) \bmod k \notin \{1, 2\}$, *then any MLAS scheduling algorithm must use a minimum of* $\left\lceil \frac{n-1}{k} \right\rceil + k$ *time slots.*

Proof. Suppose there is an algorithm that can complete the aggregation convergecast in $\lceil \frac{n-1}{k} \rceil + k - 1$ time slots. Then such an algorithm must have the following properties:

1. It needs to assign time $1, \ldots, k+1$ to nodes in C_0.

   Let $b_i = \lceil \frac{n-1-i}{k} \rceil - 1 + t_i$ be the lower bound on the time at which the data of node S_i reaches the sink. If $\exists i, 1 \leq i \leq k+1$ such that $t_i > k+1$, then

$$\begin{aligned} b_i &\geq \left\lceil \frac{n-1-(k+1)}{k} \right\rceil - 1 + (k+2) \\ &= \left\lceil \frac{n-2}{k} \right\rceil + k \\ &> \left\lceil \frac{n-1}{k} \right\rceil + k - 1 \end{aligned}$$

2. The node in C_0 that transmits at time $k+1$ has to transmit to a node in C_1. This follows from the fact that a node cannot receive data after it has transmitted its own data.

3. Nodes in C_1 must be assigned time slots from the set $\{1, \ldots, k, k+2\}$ following a similar argument as in Property 1. Time $k+1$ is excluded because a node in

C_1 is receiving data at time $k+1$ (see Property 2), and we know from Lemma 2 that no other node in C_1 can transmit at the same time.

4. The node in C_1 that transmits at time $k+2$ has to transmit to a node in C_2, following a similar argument as in Property 2.

5. Node S_{k+1} has to transmit in the first $k+1$ time slots, otherwise its data cannot reach the sink in $\lceil \frac{n-1}{k} \rceil + k - 1$.

   Proof. Suppose S_{k+1} transmits at time $k+2$. Then its data cannot reach the sink before $\lceil \frac{n-1-(k+1)}{k} \rceil - 1 + (k+2)$.

   $$\left\lceil \frac{n-1-(k+1)}{k} \right\rceil - 1 + (k+2) \leq \left\lceil \frac{n-1}{k} \right\rceil - 1 + (k+2)$$

   We substitute $n-1$ by $ik+r$.

   $$\left\lceil \frac{ik+r-k-1}{k} \right\rceil + 2 \leq \left\lceil \frac{ik+r}{k} \right\rceil$$
   $$\left\lceil \frac{r-1}{k} \right\rceil + 1 \leq \left\lceil \frac{r}{k} \right\rceil$$

   This is only true if $r = 1$. But we assumed that $(n-1) \bmod k \notin \{1, 2\}$, so S_{k+1} cannot transmit at time $k+2$ or later. ■

6. Suppose that S_{k+1} transmits at some time t with $1 \leq t \leq k+1$. By Property 1, there exists a node $S_i \in C_0$ that also transmits at time t. In order to have a collision free transmission, S_0 is the only possible receiver for S_i. S_{k+1} would interfere with any other possible receiver.

7. It has to assign a time slot $i \leq k+2$ to node S_{2k+2}. However, it cannot transmit at time $k+2$ because a node in C_1 transmits to a node in C_2 at the same time (see Property 4).

Proof. Suppose S_{2k+2} transmits at time $k+3$. Then its data cannot reach the sink before $\lceil \frac{n-1-(2k+2)}{k} \rceil - 1 + (k+3)$.

$$\left\lceil \frac{n-1-(2k+2)}{k} \right\rceil - 1 + (k+3) \leq \left\lceil \frac{n-1}{k} \right\rceil - 1 + (k+3)$$

We substitute $n-1$ by $ik+r$.

$$\left\lceil \frac{ik+r-2k-2}{k} \right\rceil + 3 \leq \left\lceil \frac{ik+r}{k} \right\rceil$$
$$\left\lceil \frac{r-2}{k} \right\rceil + 1 \leq \left\lceil \frac{r}{k} \right\rceil$$

This is only true if $r \in \{1,2\}$. But we assumed that $(n-1) \bmod k \notin \{1,2\}$, so S_{2k+2} cannot transmit at time $k+3$ or later. ■

Suppose that node S_{2k+2} is transmitting at time $i \in \{1,\ldots,k\}$, which means it is transmitting at the same time as a node $S_i \in C_1$. Then the only possible receiver for S_i is S_{k+1}. Otherwise, if the receiver is $S_{i'}$ with $i' < k+1$, then a node in C_0 will interfere at $S_{i'}$. If $i' > k+1$, then node S_{k+2} will interfere at $S_{i'}$.

If S_{k+1} is the receiver, then the only node in C_0 that can transmit at the same time is S_0. Any other node in C_0 would interfere at S_{k+1}. This implies that S_0 also has to transmit at time i and can no longer receive anything after that time. We know S_{k+1} has to transmit after time i and we know from Property 5 that it has to transmit at the same time as a node in C_0. But we know from Property 6 that the only possible receiver for the sender in C_0 is S_0. This is in contradiction with the constraint that a node cannot receive data after it has transmitted. This means that node S_{2k+2} cannot transmit at time $i \in \{1,\ldots,k\}$.

Now that we have excluded all these time slots, the only one left are $k+1$ and $k+2$. Suppose that node S_{2k+2} transmits at time $k+1$. We know from Property 2 that it

is used by a node $S_i \in C_0$ and that the receiver is $S_{i'} \in C_1$. To avoid interference at $S_{i'}$, we need to have $i' = k+1$. Therefore, S_{k+1} cannot transmit before time $k+2$, which means that its data cannot reach the sink before

$$\begin{aligned} dist(S_{k+1}, S_{n-1}) - 1 + k + 2 &= \left\lceil \frac{n-1-(k+1)}{k} \right\rceil + k + 1 \\ &= \left\lceil \frac{n-2-k}{k} \right\rceil + k + 1 \\ &= \left\lceil \frac{n-2}{k} \right\rceil + k \\ &= \left\lceil \frac{n-1}{k} \right\rceil + k \end{aligned}$$

The last equality is true because we initially assumed that $(n-1) \bmod k \neq 1$, which implies that $\lceil \frac{n-2}{k} \rceil = \lceil \frac{n-1}{k} \rceil$. This contradicts the initial assumption that the convergecast algorithm completes in $\lceil \frac{n-1}{k} \rceil + k - 1$ time slots, and thus means that node S_{2k+2} cannot transmit at time $k+1$. A similar argument can be used to show that node S_{2k+2} cannot transmit at time $k+2$.

Therefore, S_{2k+2} cannot transmit before time $k+3$, which contradicts Property 7 and completes the proof.

■

3.4.4 Algorithm for Regular Unit Interval Graphs

In this section, we introduce an algorithm for building the tree and scheduling the nodes for regular unit interval graphs. The number of time slots used by this algorithm is at worst one more than an optimal solution, and is optimal if $k \leq 2$ or $(n-1) \bmod k \notin \{1, 2\}$.

Remember that the key to any good aggregation convergecast algorithm is to maximize parallelism. To achieve that goal, our algorithm is divided in 4 main

sections that each have their own algorithm. To simplify the explanation of the algorithm, consider the nodes to be divided into groups of size k. Let us denote by G_i the group that contains nodes $S_{ik}, \ldots, S_{(i+1)k-1}$.

If $n = k + 1$, it is easy to create an optimal schedule using the Hub Algorithm, as shown in Figure 10. However, the Hub Algorithm doesn't give an optimal solution for larger networks. For instance if $n = 2k$, this approach will lead to a sub-optimal solution as shown in Figure 11. In this specific example, the lower bound given by Theorem 12 is $k + 1$. However, since nodes in G_0 transmit to S_k, nodes in G_1 cannot reuse the same time slots without causing interference at S_k. Thus, a total of $2k$ time slots are necessary when using the Hub Algorithm for this example.

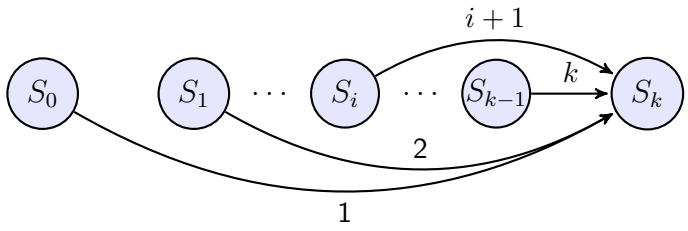

Figure 10: Optimal solution using the Hub Algorithm.

A better approach is to use node S_0 as a data aggregator for the nodes in G_0. Because S_0 is more than k nodes away from the nodes in G_1, nodes in G_1 are far enough to allow the same time slots to be reused without interference. This scenario is illustrated in Figure 12, for $n = sk$ and is an optimal schedule for $n \leq 2k$.

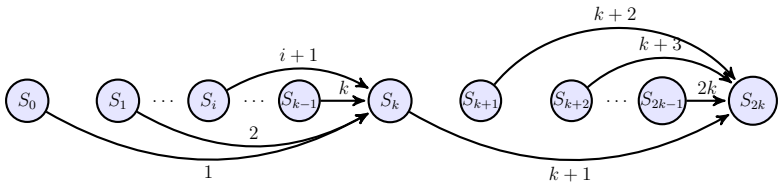

Figure 11: Illustration of the tree and schedule built by the Hub Algorithm for a Regular Unit Interval Graph of size $2k$. The schedule built with the Hub Algorithm uses $2k$ time slots whereas an optimal solution for the same graph would use only $k + 1$ time slots.

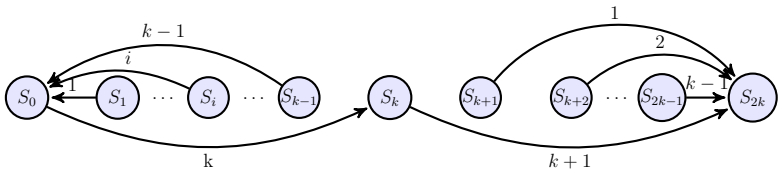

Figure 12: Illustration of using S_0 as a data aggregator for a Regular Unit Interval Graph of size $2k$. Using S_0 as the aggregator for G_0 allows us to reassign the same time slots to the nodes in G_1. In this example, we can achieve an optimal solution of $k+1$ time slots.

Things get more complicated when $n > 2k$. Nodes in G_1 now have to avoid collisions with nodes in G_2 as well as avoiding collisions with nodes in G_0. This adds constraints that make it harder to maintain an optimal solution. Nodes in G_1 cannot transmit backwards to avoid collisions with G_2, because there would be a collision with a node in G_0. In order to minimize the chance of collision, a simple solution is to schedule nodes in G_1 to transmit to the closest forward neighbor. This way we still avoid collisions with nodes in G_0 while increasing the number of time slots that can be reused in G_2 (see Figure 13).

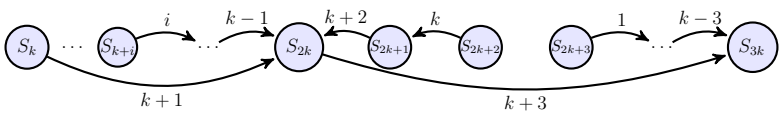

Figure 13: Tree and schedule for G_1 and G_2. Nodes in G_1 transmit to their closest forward neighbor. This allows for $k-3$ time slots to be reusable in G_2.

The next groups face the same problem as G_1, which is to try to reuse as many time slots as possible while avoiding collisions with their neighboring groups. But we cannot reuse the same scheduling strategy that we used for G_1 because that would not be collision free. In G_1, node S_{k+i} transmits to S_{k+i+1} at time i. Nodes in G_2 can only transmit at time i if they are at least $k+1$ nodes away from S_{k+i+1}. Therefore, the first node in G_2 that can transmit at time i is node S_{2k+i+2}. This means that S_{2k+1} and S_{2k+2} need to use time slots that are not used in G_1. As shown in Figure 13,

we can make them transmit backwards so that we can reuse those time slots in the next group.

Next, in G_3, we cannot reuse the exact pattern that we used in G_2 without losing optimality. We can schedule nodes $S_{3k+5}, \ldots, S_{4k-1}$ to use time slots $1, \ldots, k-5$ respectively. But that leaves us with 4 nodes that would need to transmit backwards, and we can only use 3 more time slots if we want to remain optimal. But because we chose to make nodes S_{2k+1} and S_{2k+2} transmit backwards, we can reuse one of their time slot to schedule node S_{3k+4} to transmit to S_{4k}. This solution is shown in Figure 14.

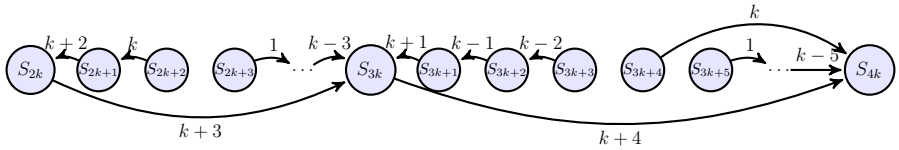

Figure 14: Tree and schedule for G_2 and G_3. Some nodes in G_2 and G_3 will transmit to their closest backward neighbor to allow for more time slots to be reused in the next groups.

We can now introduce our algorithm for constructing the tree and scheduling the nodes of a regular unit interval graph of size $n > 2k+2$. As shown in Figure 12, S_0 is used as a data aggregator for nodes in G_0 and will transmit its aggregated data to S_k at time k. Nodes in the other groups will be scheduled according to a common pattern. Let G_i be divided into 4 sub-groups A_i, B_i, C_i and D_i, and let $\alpha = min\{i, k-1\}$ and $\beta = min\{2i-2, k-1\}$. The sub-groups are defined by the

following equations

$$A_i = \{S_{ik}\}$$

$$B_i = \begin{cases} \emptyset & \text{if } i = 0 \\ \{S_j \mid ik+1 \leq j \leq ik+\alpha\} & \text{otherwise} \end{cases}$$

$$C_i = \{S_j \mid ik+\alpha+1 \leq j \leq ik+\beta\}$$

$$D_i = \{S_j \mid ik+\beta+1 \leq j \leq ik+k-1\}$$

From these equations, we can derive the size of each sub-group:

$$|A_i| = 1$$

$$|B_i| = \alpha$$

$$|C_i| = \begin{cases} \beta - \alpha & \text{if } i < k-1 \\ 0 & \text{otherwise} \end{cases}$$

$$|D_i| = \begin{cases} k-1-\beta & \text{if } i \leq \frac{k}{2} \\ 0 & \text{otherwise} \end{cases}$$

Based on these definitions, our regular unit interval graph algorithm schedules nodes in each sub-group using the following rules:

1. $S_{ik} \in A_i$ transmits at time $k+i+1$. If $ik+k < n$ then the receiver is node S_{ik+k}, otherwise the receiver is node S_{n-1}.

2. Let T_i be the set of time slots used by group G_i and let $TB_i = \{1, \ldots, min(k+i, 2k-1)\} - T_{i-1}$ be the set of time slots used by sub-group B_i. Node $S_j \in B_i$ transmits to node S_{j-1} at time $j' \in TB_i$. Time slots are assigned in ascending order from the last node to the first node in B_i.

3. For each $S_j \in C_i$, S_j transmits at the same time as node $S_{j-k-\alpha} \in B_{i-1}$. If $ik + k < n$, the receiver is node S_{ik+k}, otherwise the receiver is node S_{n-1}.

4. For each $S_j \in D_i$, S_j transmits to node S_{j+1} at time $j - (ik + \beta)$. Nodes in D_i will therefore use time slots in the range $1, \ldots, |D_i|$.

Theorem 13 *Our regular unit interval graph scheduling algorithm produces a valid collision-free schedule that uses exactly* $\left\lceil \frac{n-1}{k} \right\rceil + k$ *time slots. This schedule can be built in* $\mathcal{O}(|V|)$ *time.*

Proof. We can easily see in Figure 12 that the schedule of the first $2k$ nodes is collision-free. We will show by induction that the schedule of the other nodes is also collision-free. Assume that nodes in the first i groups have a collision-free schedule. We know from the algorithm's rules that each node in G_i is assigned a different time slot, so there cannot be any collision between them. We need to show that their transmissions never collide with the transmission of a node in G_{i-1}.

The transmission of node S_{ik} is obviously collision-free since the latest time slot used in G_{i-1} is time $k+i$ and S_{ik} transmits at time $k+i+1$. Nodes in B_i are assigned time slots from the set TB_i, which is disjoint from T_{i-1} by definition. Therefore, the schedule of nodes in B_i is also collision-free, as long as the number of time slots in TB_i is greater or equal to the number of nodes in B_i. This is verified by solving

$$\begin{aligned} |B_i| &<= |TB_i| \\ &<= min\{k+i, 2k-1\} - k \\ &= min\{i, k-1\} \\ &= \alpha \end{aligned}$$

To prove that the schedule of nodes in C_i is collision-free, we first need to prove

that $|B_{i-1}| >= |C_i|$. If $i >= k-1$, then $|C_i| = 0$ so $|B_{i-1}|$ is definitely greater than $|C_i|$. If $i < k-1$, then

$$
\begin{aligned}
|B_{i-1}| &>= |C_i| \\
min\{i-1, k-1\} &>= min\{2i-2, k-1\} - min\{i, k-1\} \\
i-1 &>= min\{2i-2, k-1\} - i \\
i-1 &>= min\{i-2, k-1-i\}
\end{aligned}
$$

This is always true since $i-1 > i-2$. Therefore, nodes in C_i will use the same time slots as nodes in B_{i-1}. This is collision-free because nodes in C_i transmit forwards while nodes in B_{i-1} transmit backwards, and because for each pair of nodes that share the same time slot, the distance between them is more than k nodes (exactly $k+\alpha$ nodes). Those conditions guarantee that the receivers will be in range of only one transmitter at their scheduled time slots.

Finally, we need to prove that the schedule of nodes in D_i is collision-free. Observe that time slots used by D_i are a strict subset of time slots used by D_{i-1}. Therefore, nodes in D_i can only collide with node in D_{i-1}. Suppose that $S_j \in D_i$ transmits to S_{j+1} and that $S_{j'} \in D_{i-1}$ transmits to $S_{j'+1}$, both at time $j-(ik+\beta)$. We need to verify that the distance between S_j and $S_{j'+1}$ as well as the distance between $S_{j'}$ and S_{j+1} are both greater than k. Based on the definition of our algorithm, we can represent j' in terms of j and find that $j' = j-k-2$. We can then compute both distances and find that the distance between S_j and $S_{j'+1}$ is $j-(j-k-1) = k+1$ and that the distance between $S_{j'}$ and S_{j+1} is $(j+1)-(j-k-2) = k+3$. Both distances are greater than k and the schedule is therefore collision-free.

Now that we have proven that the algorithm is collision-free, we need to prove that the schedule uses exactly $\lceil \frac{n-1}{k} \rceil + k$ time slots. By the definition of the algorithm,

the highest time slot used by G_i is $k+i+1$, so the last time slot will be used by the last group. There are $\lceil \frac{n-1}{k} \rceil$ groups, numbered from 0 to $\lceil \frac{n-1}{k} \rceil - 1$. Thus, the last time slot used by the last group will be $k + (\lceil \frac{n-1}{k} \rceil - 1) + 1 = \lceil \frac{n-1}{k} \rceil + k$.

As for the Hub Algorithm for unit interval graphs, we start by dividing the graph into cliques which can be done in $\mathcal{O}(|V|)$ time. The division of each clique into subgroups can be computed in constant time. It is straightforward to see from the description of the schedule that the remaining computations can be completed in $\mathcal{O}(|V|)$ time. ■

Chapter 4

Heuristics for Arbitrary Graphs

In this chapter, we present a new algorithm for building an aggregation tree and two new scheduling algorithms for the MLAS problem. Section 4.1 describes our tree-building algorithm, called Degree-Constrained Aggregation Tree (DCAT), and presents a simple example showing how it can reduce the latency. Our scheduling algorithms, named WIRES-G (where G stands for Greedy) and Degree-Constrained Aggregation Tree Scheduling (DCATS), are presented in Section 4.2. In Section 4.3, the performance of DCAT is evaluated through simulations on randomly generated graphs. The same randomly generated graphs are used in Section 4.4 to evaluate the performance of WIRES-G and DCATS. An in-depth analysis of the results for the tree-building algorithm and the scheduling algorithms is done in Section 4.5.

4.1 Degree-Constrained Aggregation Tree (DCAT)

As mentioned in Chapter 2, most approximation algorithms designed for any topologies use either a CDS-based (or MIS-based) approach or they use a Shortest Path Tree (SPT) algorithm. It was shown by Malhotra et al. [30] that SPT-based approaches performed better in practice due to the fact that CDS approaches tend to

cluster nodes, which reduces the possibilities of scheduling many nodes in parallel. It was also shown in [30] that the choice of a good tree was very important to produce a good schedule, and they proposed the BSPT algorithm to build an efficient aggregation tree. The BSPT algorithm tries to maximize parallelism by distributing evenly the nodes between their potential parents. The main idea behind it is that if nodes are distributed evenly, the scheduling algorithm is more likely to be able to schedule many nodes in parallel.

However, the approach used in BSPT doesn't take into consideration the degree of the nodes in the graph. So although BSPT does a good job of minimizing the maximum degree of a node in the aggregation tree, it does little to prevent a high-degree node in the graph from having many children in the tree. Having a high-degree node with multiple children reduces the possibility of parallelism, and in the end hurts the overall performance of the scheduling algorithm.

Our new algorithm DCAT tries to address this problem by minimizing the number of children assigned to the highest-degree nodes in the graph. It works by traversing the graph in a BFS way. As it traverses each node, the set of potential parents is determined by identifying the nodes that are one-hop closer to the sink. The potential parent with the lowest degree in the graph is selected as the parent for the currently traversed node. As with any BFS-based algorithm, DCAT has a time complexity of $\mathcal{O}(|V| + |E|)$ since every vertex and every edge are explored once.

Figure 15 illustrates how the approach used in DCAT can give better results than BSPT. It shows a case where distributing the children evenly between potential parents, regardless of their degree in the graph, leads to more constraints for the scheduling algorithm. In this specific example, we can clearly see that considering the degree in the graph is better than trying to minimize the degree in the aggregation tree. This gives an indication that DCAT could be a better algorithm for building an

Algorithm 4 DCAT

Input: $G = (V, E)$, s: sink node
Output: A spanning tree of G rooted at s where $v.p$ is the parent of $v \in V$

```
1: procedure DCAT(G, s)
2:     for each u ∈ G.V do
3:         u.d = −1
4:         u.p = nil
5:     end for
6:     s.d = 0
7:     Q = ∅
8:     ENQUEUE(Q, s)
9:     while Q ≠ ∅ do
10:        u = DEQUEUE(Q)
11:        for each v ∈ 𝒩(u) do
12:            if v.d < 0 then
13:                v.d = u.d + 1
14:                ENQUEUE(Q, v)
15:            end if
16:            if v.d > u.d then
17:                if v.p == nil or |𝒩(v.p)| > |𝒩(u)| then
18:                    v.p = u
19:                end if
20:            end if
21:        end for
22:    end while
23: end procedure
```

efficient aggregation tree, at least when the degree varies between nodes. In order to evaluate the performance of DCAT and validate the potential gains, we conducted a series of experiments that are presented in Section 4.3.

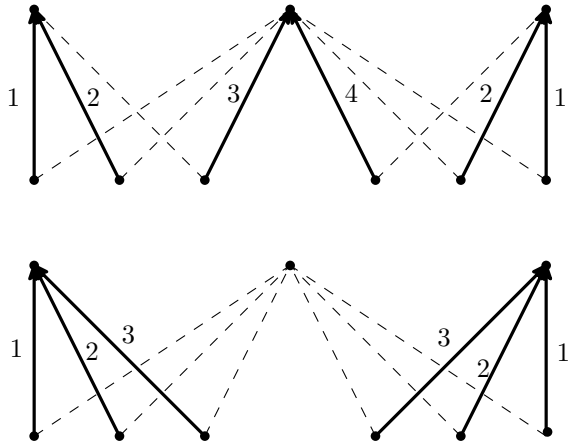

Figure 15: Example showing that minimizing the highest degree in the aggregation tree is not necessarily a good approach. Solid edges: Aggregation tree edges. Dashed edges: Graph edges that are not tree edges. Above: Aggregation tree built by the BSPT algorithm. We can see that the best possible schedule for this tree requires 4 time slots. Below: Aggregation tree built by the DCAT algorithm for the same graph. In this case, the best schedule uses only 3 time slots.

4.2 Scheduling Algorithms

In this section, we present two new scheduling algorithms based on the WIRES algorithm designed by Malhotra et al. [30]. Those new algorithms are the result of the analysis of our simulation results in section 4.5.1. As our analysis shows, the DCAT algorithm sometimes assigns many children to the same parent in the aggregation tree. This has an impact on the latency of any scheduling algorithm that preserves the original aggregation tree. To alleviate this problem, we added a new step to the WIRES scheduling algorithm to try to schedule more nodes per time slot. At each round, the new step tries to find alternate parents for the eligible nodes that

could not be scheduled by the original algorithm.

Our first scheduling algorithm is called WIRES-G (G for Greedy) and is detailed in Algorithm 5. It works as follows. Lines 5-9 correspond to the steps that are done at each round in the original WIRES algorithm. At the end of Line 9, S contains the nodes that are scheduled to send at time j and R contains the set of nodes that receive data from the nodes in S. At this point, if we keep the original tree, no other node can be scheduled to transmit without causing interference with the already scheduled nodes. Our experiments have shown that in many cases, more nodes have the potential to be scheduled if we just select a new parent for them. This is especially true at high densities where each node have many potential parents. This is what our new Greedy-Scheduling procedure tries to accomplish.

The loop of lines 16-35 iterates through all the eligible nodes that were not scheduled to transmit. Line 17 first verifies if the current node is not already a receiver and if it can transmit without causing any interference at one of the receivers in R. If the node p can transmit without conflict, the loop of lines 19-23 iterates through all the neighbors of p and looks for a neighbor that has not yet been scheduled and that can receive at time j without conflict. If more than one neighbor follow these constraints, the one with the lowest degree in the graph will be selected. If a neighbor is found, it becomes the new parent in the aggregation tree for the current node. If the previous parent does not have any unscheduled children left and if it is not a receiver in the current round, it is added to the list of eligible nodes as it could potentially be scheduled at time j (lines 27-29).

GetEligibleNodes takes $\mathcal{O}(|V|)$ time and ComputeWeights takes $\mathcal{O}(|V| + |E|)$ time. Line 7 takes $\mathcal{O}(|V \log |V||)$ time in the worst case. Both the ScheduleNodes procedure and the Greedy-Scheduling procedure explore all nodes and edges at each call and have a time complexity of $\mathcal{O}(|V| + |E|)$. The outer loop of

Algorithm 5 WIRES-G

Input: $G = (V, E)$, s: sink node, $v.p$: parent of $v \in V$ in the tree
Output: A valid schedule for G where $v.t$ is the transmission time for $v \in V$

1: **procedure** WIRES-G(G, s)
2: $\forall v \in G.V$ $v.t = 0$ ▷ Initialize time slots
3: $j = 1$
4: **while** $s.t = 0$ **do**
5: $L =$ GetEligibleNodes(G)
6: ComputeWeights(L)
7: SortDecreasing(L) ▷ Sort nodes in decreasing order of weights.
8: $S = R = \emptyset$ ▷ Set of senders and receivers are initially empty
9: ScheduleNodes(L, S, R)
10: $L = L \setminus S$ ▷ Remove senders from eligible nodes
11: Greedy-Scheduling(L, S, R)
12: $j = j + 1$
13: **end while**
14: **end procedure**

15: **procedure** Greedy-Scheduling(L, S, R)
16: **for each** $u \in L$ **do**
17: **if** $u \notin R$ and $u \notin \mathcal{N}(R)$ **then** ▷ If u can transmit without interference
18: $r = nil$ ▷ No parent found yet
19: **for each** $p \in \mathcal{N}(u)$ **do** ▷ Try to find a parent
20: **if** $p.t == 0$ and $p \notin \mathcal{N}(S)$ and ($r = nil$ or $|\mathcal{N}(p)| < |\mathcal{N}(r)|$) **then**
21: $r = p$ ▷ Found a parent with no conflict
22: **end if**
23: **end for**
24: **if** $r \neq nil$ **then** ▷ If a new parent was found
25: $p = u.p$ ▷ Keep a reference to the previous parent
26: $u.p = r$ ▷ Assign the new parent
27: **if** IsEligible(p) **then** ▷ Check if p has become eligible.
28: $L = L \cup \{p\}$ ▷ Add it to the set of eligible nodes
29: **end if**
30: $u.t = j$ ▷ u is scheduled to transmit at time j
31: $S = S \cup \{u\}$ ▷ u is added to the set of senders
32: $R = R \cup \{u.p\}$ ▷ The parent of u is added to the set of receivers
33: **end if**
34: **end if**
35: **end for**
36: **end procedure**

line 4 executes at most $|V|$ times. Thus, the WIRES-G algorithm, like the WIRES algorithm, has time complexity $\mathcal{O}(|V|^2 \log |V| + |V||E|)$.

The WIRES-G scheduling algorithm was shown to obtain very good results when used in combination with either DCAT or BSPT (see Section 4.4). However, because it starts by scheduling as many nodes as possible using the original tree, its performance is limited by the given tree. Another approach would be to completely get rid of the scheduling routine in the original WIRES and use only the Greedy-Scheduling procedure introduced in WIRES-G. This way, the impact of the original tree would be more limited and the overall performance could be improved. This is the idea that is used in our DCATS algorithm. DCATS combines our tree-building algorithm DCAT with our Greedy-Scheduling procedure to try to achieve even lower latency.

Algorithm 6 DCATS

Input: $G = (V, E)$, s: sink node
Output: A spanning tree of G rooted at s and a valid schedule for G, where $v.t$ and $v.p$ are the transmission time and parent for $v \in V$ respectively.

```
1:  procedure DCATS(G)
2:      DCAT(G, s)
3:      ∀v ∈ G.V v.t = 0                          ▷ Initialize time slots
4:      j = 1
5:      while s.t = 0 do
6:          L = GetEligibleNodes(G)
7:          ComputeWeights(L)
8:          SortDecreasing(L)          ▷ Sort nodes in decreasing order of weights.
9:          S = R = ∅                  ▷ Set of senders and receivers are initially empty
10:         Greedy-Scheduling(L, S, R)
11:         j = j + 1
12:     end while
13: end procedure
```

The pseudocode for DCATS is shown in Algorithm 6. DCATS first uses the DCAT algorithm to build an aggregation tree. This is necessary as the original tree will determine the order in which the nodes will be considered by the scheduling

phase of the algorithm. The scheduling phase in DCATS (Lines 5-12) is similar to the scheduling done in WIRES. At each iteration, we determine which nodes are eligible to be scheduled at this round. Nodes become eligible once all their children have been assigned a time slot. DCATS uses the same weight calculation used in WIRES (number of non-leaf neighbors) and nodes are sorted in decreasing order of weight. The GREEDY-SCHEDULING procedure then iterates through all the eligible nodes and tries to schedule as many as possible without conflict. Obviously, the node with the highest weight is always scheduled for the current round, and the lower weight nodes are only scheduled if doing so will not cause any conflict with the previously scheduled nodes.

DCATS differs from WIRES-G in two ways: (a) it uses DCAT to build a tree and (b) there is no call to the SCHEDULENODES procedure. As already mentioned, DCAT takes time $\mathcal{O}(|V| + |E|)$ and WIRES-G takes time $\mathcal{O}(|V|^2 \log |V| + |V||E|)$. Therefore, it is easy to see that DCATS has time complexity $\mathcal{O}(|V|^2 \log |V| + |V||E|)$.

4.3 Simulation Results for DCAT

In this section, we evaluate the performance of the DCAT algorithm on randomly generated graphs. Because of the fact that DCAT only solves the tree-building part of the MLAS problem, we pair it with the WIRES scheduling algorithm presented in [30] to generate the schedule. We selected the WIRES algorithm for its ability to retain the original aggregation tree. Moreover, it was shown in [30] to be the best among the scheduling algorithms that don't alter the aggregation tree. The DCAT-WIRES combination is compared to the BSPT-WIRES pair, which was the best combination in the simulations conducted in [30]. We measure the latency of the generated schedule, which corresponds to the total number of time slots required

by the schedule.

4.3.1 Performance Comparison for Small Graphs (5 by 5)

We look at the performance of the DCAT algorithm on small graphs. The test graphs were generated by uniformly distributing the nodes at random in a geographic area of 5 by 5. Nodes have a transmission range of 1 and the graphs are generated with a wide range of densities between 8 and 200, where the density is the average degree in the graph. Nodes are connected in the generated graph if the Euclidean distance between them is less than or equal to the transmission range. For each selected density, a total of 100 connected graphs were generated and the results are averaged over these 100 graphs.

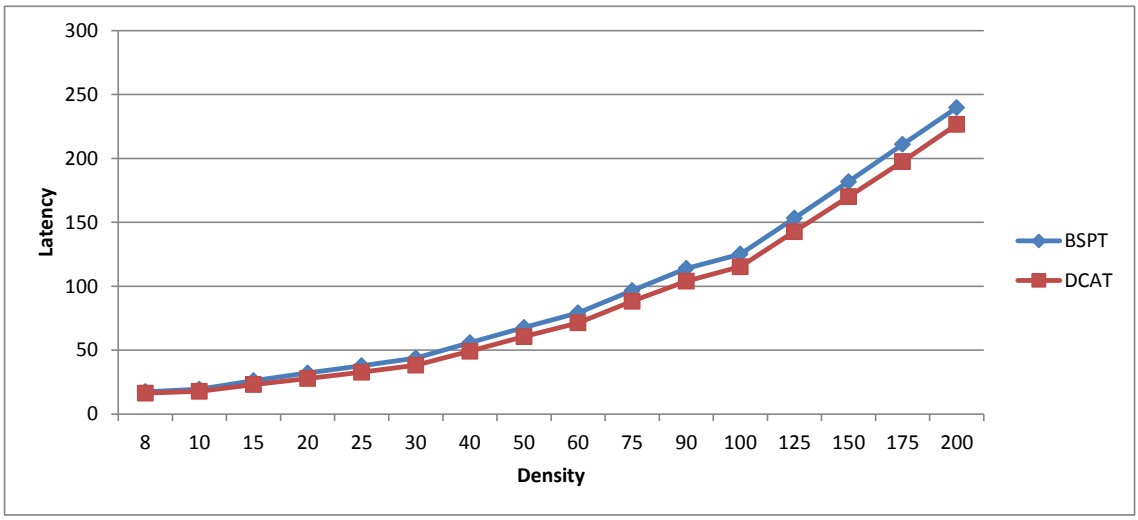

Figure 16: Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 5 by 5.

As shown in Figure 16, the DCAT algorithm is better at all densities. As the density increases, so does the improvement in the schedule length. This can be easily explained by the fact the the BSPT algorithm does not consider the nodes' degree at all when building the aggregation tree. Thus, it is outperformed at high densities

by an algorithm that tries to avoid selecting high-degree nodes as parents for other nodes.

# Nodes	Density	BSPT	DCAT	Difference	Gain (%)
80	8	17.45	16.29	1.16	6.65
96	10	19.46	17.74	1.72	8.84
145	15	26.08	22.99	3.09	11.85
192	20	32.11	27.82	4.29	13.36
235	25	37.72	32.71	5.01	13.28
284	30	43.73	38.2	5.53	12.65
380	40	55.89	49.15	6.74	12.06
475	50	67.78	60.6	7.18	10.59
568	60	79.11	71.29	7.82	9.88
712	75	96.58	88.2	8.38	8.68
855	90	113.96	103.94	10.02	8.79
950	100	125.19	115.26	9.93	7.93
1185	125	153.2	142.7	10.5	6.85
1420	150	181.78	170.07	11.71	6.44
1662	175	210.89	197.55	13.34	6.33
1900	200	239.53	226.48	13.05	5.45

Table 2: Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 5 by 5.

Figure 17 shows the performance difference in percentage. We can see that the gains in percentage are bigger as the density increases up until it reaches a peak of $\sim$13% at a density of 20. The gains in percentage start to slowly decrease after that peak, but they remain above 5% at all densities. The decrease in gains can be explained by the fact that at a certain point, no matter which node we select as the parent, it still has a high degree and the scheduling algorithm cannot parallelize as many transmissions as it can at lower densities. The fact that the graph is relatively small further complicates matters as many nodes compete for the same channel of communication in a small area. This reduces the number of nodes that can be scheduled in parallel.

Out of the 1600 graphs that were generated for these tests, the BSPT algorithm

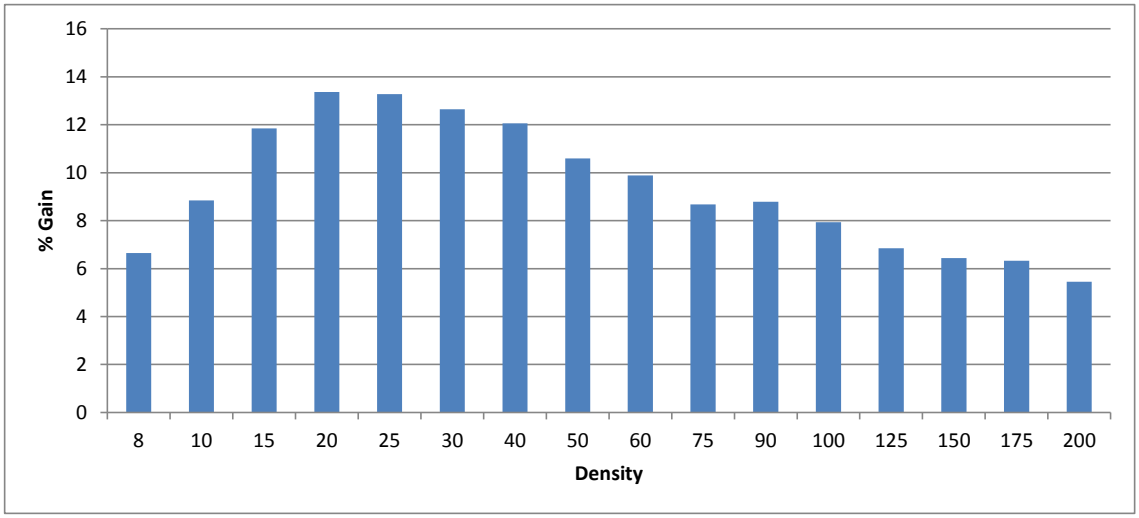

Figure 17: Average gains of using the DCAT algorithm when compared to the BSPT algorithm for a network size of 5 by 5.

was only able to outperform the DCAT algorithm in 78 instances, or a little less than 5%. Interestingly, 42 of those cases occurred at a density of 125 or above. This indicates that further improvements could be made at high densities to address those cases.

4.3.2 Performance Comparison for Medium-Sized Graphs (10 by 10)

We look at the performance of the DCAT algorithm on medium-sized graphs, generated by uniformly distributing the nodes at random in a geographic area of 10 by 10. The rest of the setup is the same as the previous subsection with nodes having a transmission range of 1 and the graphs having a density between 8 and 200. For each selected density, a total of 100 connected graphs were generated and the results are averaged over these 100 graphs.

As was the case for small graphs, the DCAT algorithm beats the BSPT algorithm at all densities and for almost all test graphs. We can see in Figure 18 that the

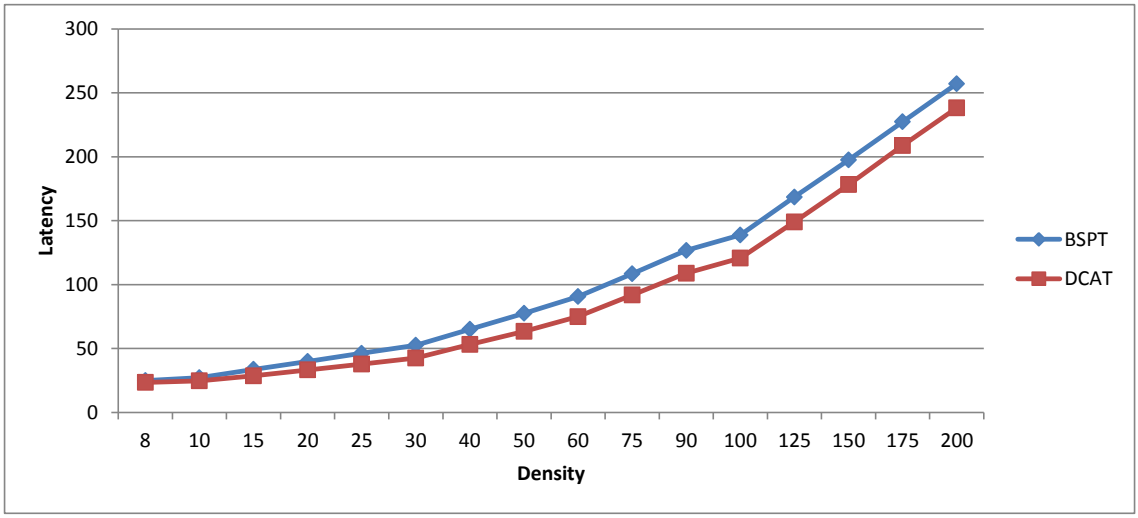

Figure 18: Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 10 by 10.

gains in number of time slots are bigger at higher densities. This time, the gap between DCAT and BSPT is a little bigger than with smaller graphs, which means that the higher number of nodes seems to benefit our algorithm. This was somewhat expected since a bigger graph gives more possibilities for parallelism, and the way DCAT selects parent also improves the likelihood that the scheduling algorithm will be able to schedule multiple nodes at the same time slot.

The BSPT algorithm was only the best in 23 out of the 1600 medium-sized graphs generated, for a meager 1.4%. Again, most of those cases occur at high densities with 17 at a density of 150 or above. In one specific case that occurred at a density of 200, the result of the DCAT algorithm is particularly bad when compared with the result of BSPT. Indeed, the BSPT algorithm produces a schedule that is 14.95% better than DCAT, by far the worse performance in any of our tests. This suggests that there might be some rare cases in which the DCAT algorithm is not a good approach.

Figure 19 presents the gains in percentage for the medium-sized graphs. The chart follows a similar pattern as Figure 17, where gains are bigger as the density increases

# Nodes	Density	BSPT	DCAT	Difference	Gain (%)
280	8	24.96	23.47	1.49	5.97
350	10	27.22	24.78	2.44	8.96
520	15	33.62	28.65	4.97	14.78
690	20	39.91	33.22	6.69	16.76
870	25	46.24	37.88	8.36	18.08
1040	30	52.5	42.56	9.94	18.93
1385	40	64.97	53.15	11.82	18.19
1730	50	77.49	63.45	14.04	18.12
2090	60	90.65	74.93	15.72	17.34
2610	75	108.42	91.86	16.56	15.27
3135	90	126.65	108.92	17.73	14.00
3485	100	138.71	120.77	17.94	12.93
4350	125	168.45	149.03	19.42	11.53
5220	150	197.5	178.19	19.31	9.78
6080	175	227.33	208.79	18.54	8.16
6950	200	257.08	238.26	18.82	7.32

Table 3: Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 10 by 10.

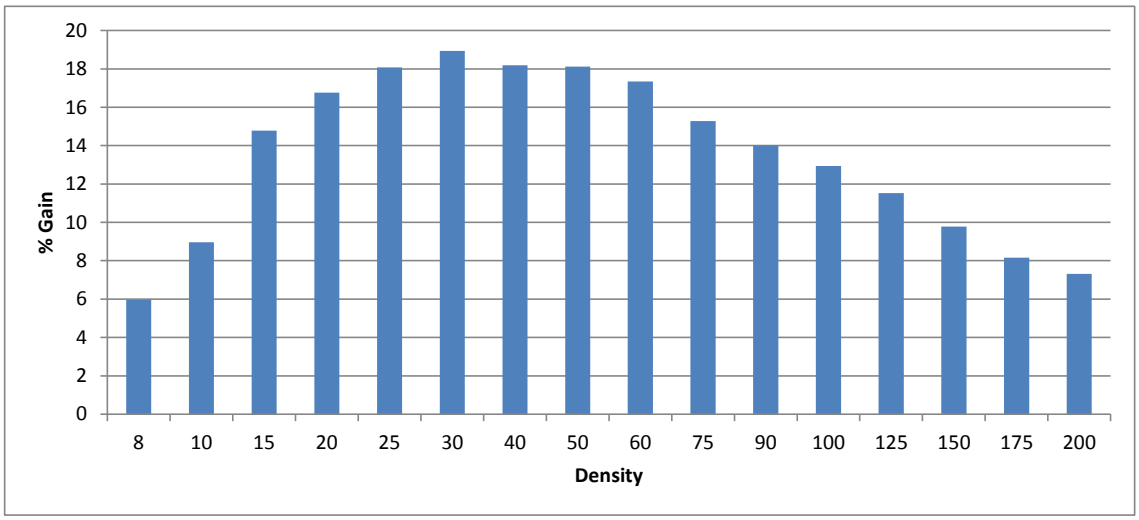

Figure 19: Average gains of using the DCAT algorithm when compared to the BSPT algorithm for a network size of 10 by 10.

up until it reaches a peak. This time, the peak occurs at a slightly higher density of 30 instead of 20. The gains in percentage are also higher with a peak of almost 19% and most of the densities are above the 10% mark. These results suggest that we could have even better gains for even bigger graphs, which is what we will try to measure in the next subsection.

4.3.3 Performance Comparison for Large Graphs (20 by 20)

We look at the performance of the DCAT algorithm on large graphs, generated by uniformly distributing the nodes at random in a geographic area of 20 by 20. The rest of the setup is the same as the two previous subsections with nodes having a transmission range of 1 and the graphs having a density between 8 and 200. For each selected density, a total of 100 connected graphs were generated and the results are averaged over these 100 graphs.

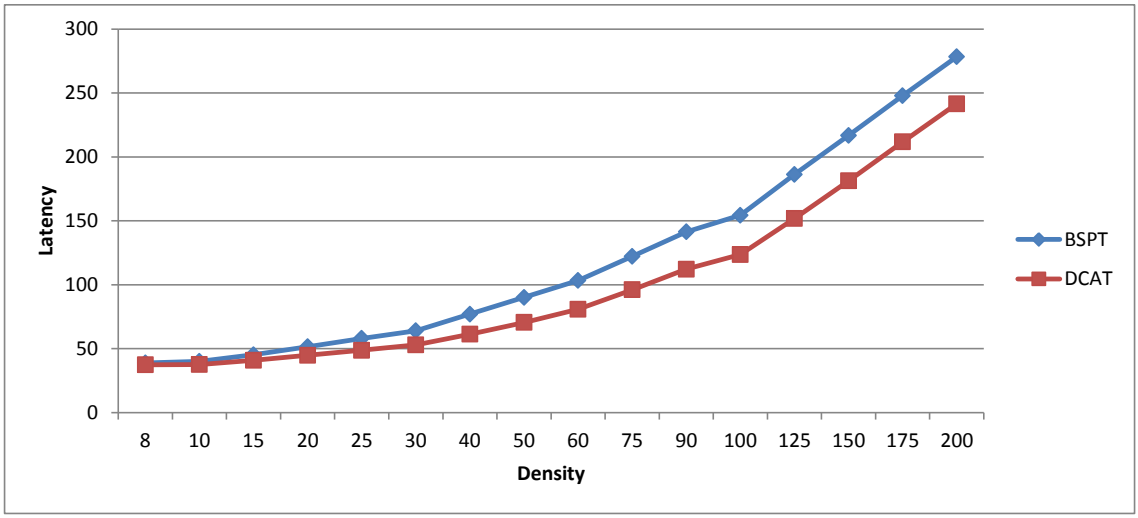

Figure 20: Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 20 by 20.

The tendency continues with the large graphs as the DCAT algorithm outperforms the BSPT algorithm at all the tested densities. The assumption that DCAT has a

bigger advantage on larger graphs is confirmed by those results, as the gap in the latency is bigger than with the two smaller sets of graphs. Figure 20 shows the same pattern that we've seen in the previous subsections where the gap in latency is bigger at higher densities. Only 6 of the 1600 graphs give the advantage to the BSPT algorithm, or 0.375% of the test cases. The worst case occurs at a density of 175 where the BSPT outperforms the DCAT algorithm by 6.02%.

# Nodes	Density	BSPT	DCAT	Difference	Gain (%)
1050	8	38.92	37.33	1.59	4.09
1335	10	40.03	37.53	2.5	6.25
2000	15	45.22	40.94	4.28	9.46
2670	20	51.41	44.77	6.64	12.92
3340	25	58.04	48.74	9.3	16.02
4005	30	63.98	53	10.98	17.16
5340	40	76.97	61.35	15.62	20.29
6650	50	90.08	70.47	19.61	21.77
7990	60	103.21	80.77	22.44	21.74
9980	75	122.12	96.05	26.07	21.35
11960	90	141.39	112.18	29.21	20.66
13300	100	154.34	123.64	30.7	19.89
16625	125	186.22	151.71	34.51	18.53
19950	150	216.69	181.28	35.41	16.34
23275	175	247.88	211.81	36.07	14.55
26600	200	278.35	241.51	36.84	13.24

Table 4: Average aggregation convergecast latency comparison between DCAT and BSPT for a network size of 20 by 20.

Figure 21 presents the gains in percentage for the large graphs. Again, gains are bigger as the density increases up until a peak is reached. This time, the peak occurs at an even higher density of 50. We also get the highest gains in percentage of all the tests with a peak of 21.77%, which is a significant gain over BSPT.

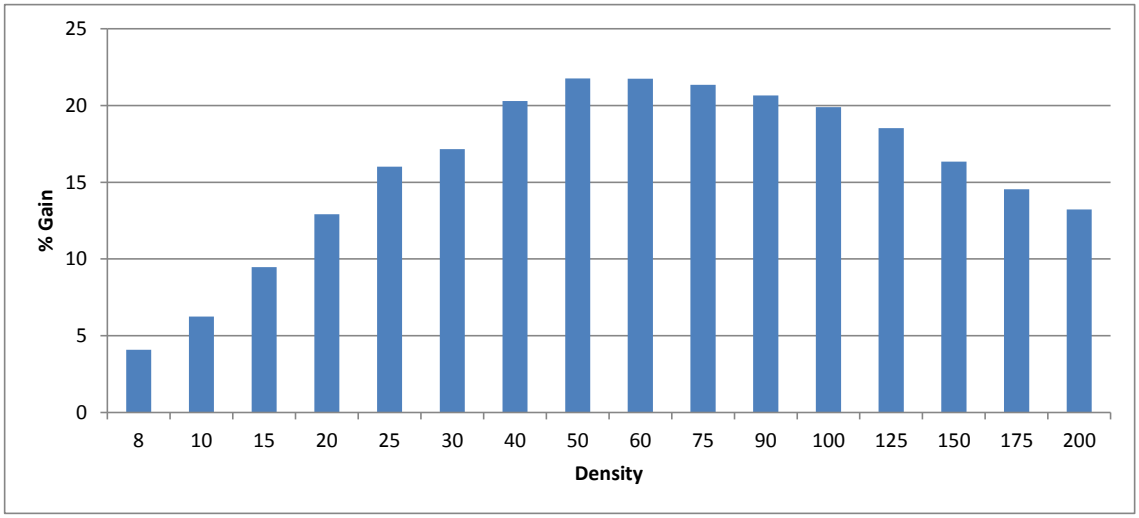

Figure 21: Average gains of using the DCAT algorithm when compared to the BSPT algorithm for a network size of 20 by 20.

4.4 Simulation Results for Scheduling Algorithms

In this section, we evaluate the performance of the WIRES-G and DCATS algorithms on randomly generated graphs. We use the same graphs that we used in Section 4.3 with the same three sizes (small, medium and large).

4.4.1 Performance Comparison for Small Graphs (5 by 5)

We start with the performance on small graphs. Figure 22 shows the performance of WIRES-G and DCATS compared with WIRES. We can see that WIRES-G reduces the latency of the schedule in all cases, regardless of the tree building algorithm used. An interesting result is that BSPT-WIRES-G is able to slightly outperform DCAT-WIRES, which suggest that the scheduling algorithm is more important that the tree-building algorithm for small graphs. The DCAT-WIRES-G algorithm beats all the other algorithms except DCATS, with bigger gains at higher densities. Figure 22 also shows the lower bounds of the shortest path trees. The lower bound is calculated using our optimal algorithm for scheduling a tree.

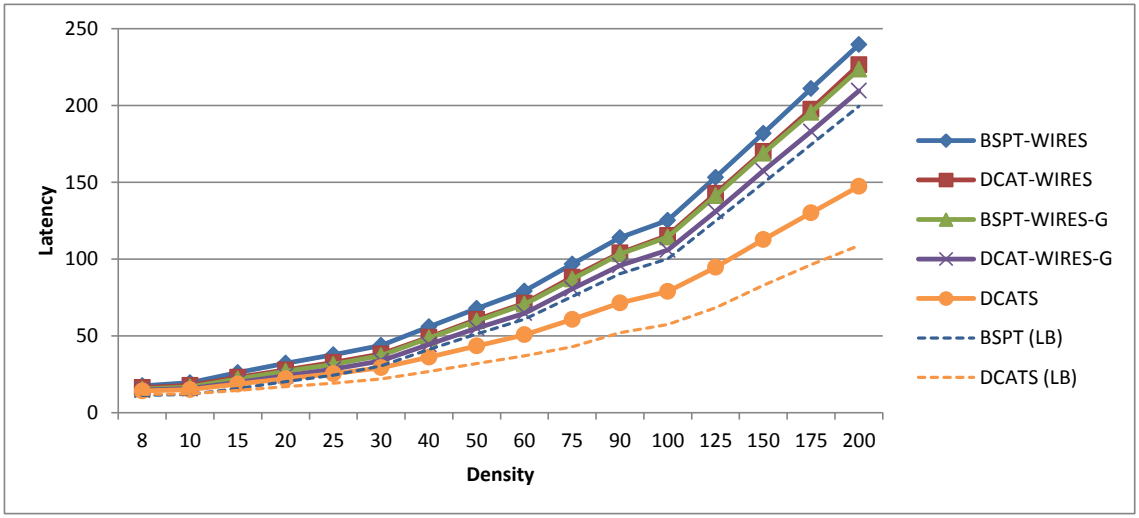

Figure 22: Average aggregation convergecast latency for a network size of 5 by 5.

Density	BSPT-WIRES	DCAT-WIRES	BSPT-WIRES-G	DCAT-WIRES-G	DCATS
8	17.45	16.29	15.36	14.64	**14.12**
10	19.46	17.74	16.78	15.61	**15**
15	26.08	22.99	21.98	19.75	**18.59**
20	32.11	27.82	26.9	24.04	**22.12**
25	37.72	32.71	31.39	28.34	**25.36**
30	43.73	38.2	37.14	33.67	**29.23**
40	55.89	49.15	48.63	44.34	**36.13**
50	67.78	60.6	59.7	54.89	**43.36**
60	79.11	71.29	70.52	64.72	**50.7**
75	96.58	88.2	86.85	80.63	**60.69**
90	113.96	103.94	103.24	95.82	**71.36**
100	125.19	115.26	114.3	105.8	**78.88**
125	153.2	142.7	141.16	131	**94.57**
150	181.78	170.07	168.84	157.35	**112.61**
175	210.89	197.55	195.4	183	**130.13**
200	239.53	226.48	223.53	209.54	**147.28**

Table 5: Average aggregation convergecast latency for a network size of 5 by 5.

In our experiments. we observed that the lower bounds of the trees generated by either BSPT, DCAT, BSPT-WIRES-G or DCAT-WIRES-G are all very similar and vary only by a latency of less than 0.5 time slots in the worst case. For this reason, we only show the lower bound of the tree generated by BSPT (usually the lowest), but the other lower bounds would appear almost identical on the chart. Note that the DCAT-WIRES-G algorithm comes close to the theoretical minimum allowed by its tree. This means that no significant gains can be achieved without further modifying the tree.

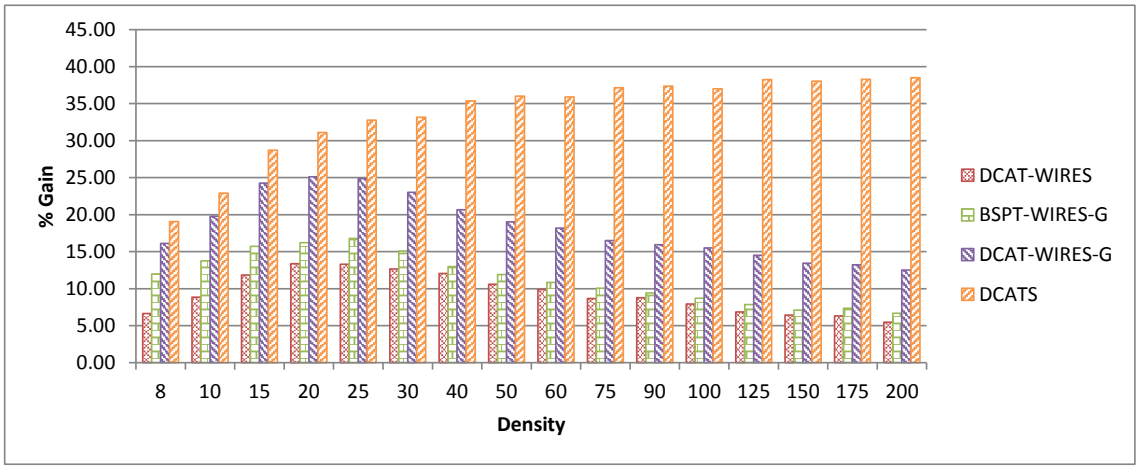

Figure 23: Average gains of using DCAT, WIRES-G or DCATS when compared to WIRES-BSPT for a network size of 5 by 5.

The best algorithm for small graphs is definitely DCATS. We can see that the average latency of the schedules produced by DCATS is way lower than the other algorithms, especially at high densities. DCATS is even significantly below the lower bound of the tree generated by BSPT. At a density of 200, DCATS outperforms DCAT-WIRES-G by more than 25%, and it beats BSPT-WIRES by almost 40%. What is even more impressive is that the gains in percentage seem to steadily increase with the density, never reaching a peak (see Figure 23).

Density	DCAT-WIRES	BSPT-WIRES-G	DCAT-WIRES-G	DCATS
8	6.65	11.98	16.10	**19.08**
10	8.84	13.77	19.78	**22.92**
15	11.85	15.72	24.27	**28.72**
20	13.36	16.23	25.13	**31.11**
25	13.28	16.78	24.87	**32.77**
30	12.65	15.07	23.00	**33.16**
40	12.06	12.99	20.67	**35.36**
50	10.59	11.92	19.02	**36.03**
60	9.88	10.86	18.19	**35.91**
75	8.68	10.07	16.51	**37.16**
90	8.79	9.41	15.92	**37.38**
100	7.93	8.70	15.49	**36.99**
125	6.85	7.86	14.49	**38.27**
150	6.44	7.12	13.44	**38.05**
175	6.33	7.35	13.22	**38.29**
200	5.45	6.68	12.52	**38.51**

Table 6: Average latency gains of using DCAT, WIRES-G or DCATS when compared to BSPT-WIRES for a network size of 5 by 5.

4.4.2 Performance Comparison for Medium-Sized Graphs (10 by 10)

We look at the performance of WIRES-G and DCATS on medium-sized graphs. As shown in Figure 24, WIRES-G again reduces the latency of the schedule in all cases when compared to WIRES. This time though, DCAT-WIRES is able to beat BSPT-WIRES-G at densities between 15 and 60 inclusively (see Table 7). This confirms what we had seen in Section 4.3 that the tree selection is more important for larger graphs. As was the case for smaller graphs, the DCAT-WIRES-G algorithm beats all the other algorithms except DCATS, and it comes close to the lower bound of the generated tree. This seems to confirm that the gains of DCAT and WIRES-G are additive.

Again, we can see that DCATS is the best at all densities and that the margin is bigger at higher densities. However, the difference with DCAT-WIRES-G is smaller

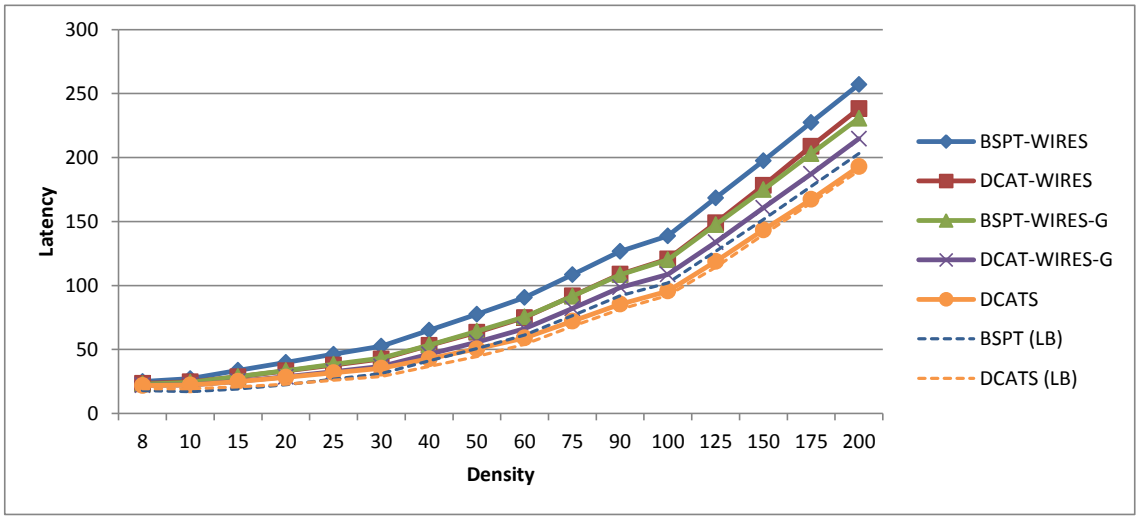

Figure 24: Average aggregation convergecast latency for a network size of 10 by 10.

Density	BSPT-WIRES	DCAT-WIRES	BSPT-WIRES-G	DCAT-WIRES-G	DCATS
8	24.96	23.47	22.96	21.95	**21.74**
10	27.22	24.78	24.31	22.32	**22.18**
15	33.62	28.65	28.82	25.23	**25.03**
20	39.91	33.22	33.46	28.67	**28.31**
25	46.24	37.88	38.37	32.7	**31.76**
30	52.5	42.56	43.22	36.69	**35.29**
40	64.97	53.15	53.49	46.27	**42.77**
50	77.49	63.45	64.2	55.69	**50.49**
60	90.65	74.93	75.36	66.25	**59.03**
75	108.42	91.86	91.62	82.11	**72.09**
90	126.65	108.92	108.42	98.54	**85.45**
100	138.71	120.77	120.06	108.72	**95.64**
125	168.45	149.03	147.51	133.84	**118.79**
150	197.5	178.19	174.95	160.7	**143.48**
175	227.33	208.79	202.97	187.22	**167.28**
200	257.08	238.26	230.68	214.75	**192.88**

Table 7: Average aggregation convergecast latency for a network size of 10 by 10.

than with smaller graphs. At densities below 25, the difference between the two is lower than 1%, which is marginal. The difference if much more impressive at higher densities where DCATS has a performance advantage of close to 10%. The performance benefits of DCATS when compared to BSPT-WIRES seem to reach a peak close to 35% at a density of 60 (see Figure 25). This is a little lower than the 38.5% obtained for smaller graphs.

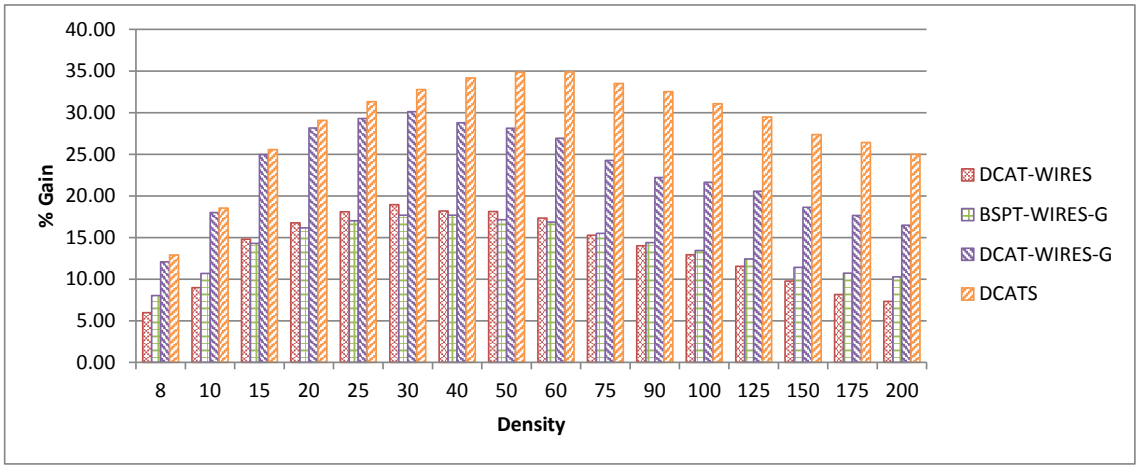

Figure 25: Average gains of using DCAT, WIRES-G or DCATS when compared to WIRES-BSPT for a network size of 10 by 10.

Note that the lower bound of the tree generated by DCATS is higher than the lower bound of the tree generated by BSPT at low densities (20 and below). This doesn't prevent DCATS from doing well at low densities as it performs close to its lower bound at every densities. It is also worth noting that DCATS outperformed BSPT-WIRES in every single graph that we tested by a minimum of 3.33%.

4.4.3 Performance Comparison for Large Graphs (20 by 20)

We look at the performance of WIRES-G and DCATS on large graphs. As shown in Figure 26, WIRES-G again reduces the latency of the schedule in all cases when compared to WIRES. However, DCAT-WIRES shows a better performance than

Density	DCAT-WIRES	BSPT-WIRES-G	DCAT-WIRES-G	DCATS
8	5.97	8.01	12.06	**12.90**
10	8.96	10.69	18.00	**18.52**
15	14.78	14.28	24.96	**25.55**
20	16.76	16.16	28.16	**29.07**
25	18.08	17.02	29.28	**31.31**
30	18.93	17.68	30.11	**32.78**
40	18.19	17.67	28.78	**34.17**
50	18.12	17.15	28.13	**34.84**
60	17.34	16.87	26.92	**34.88**
75	15.27	15.50	24.27	**33.51**
90	14.00	14.39	22.20	**32.53**
100	12.93	13.45	21.62	**31.05**
125	11.53	12.43	20.55	**29.48**
150	9.78	11.42	18.63	**27.35**
175	8.16	10.72	17.64	**26.42**
200	7.32	10.27	16.47	**24.97**

Table 8: Average latency gains of using DCAT, WIRES-G or DCATS when compared to BSPT-WIRES for a network size of 10 by 10.

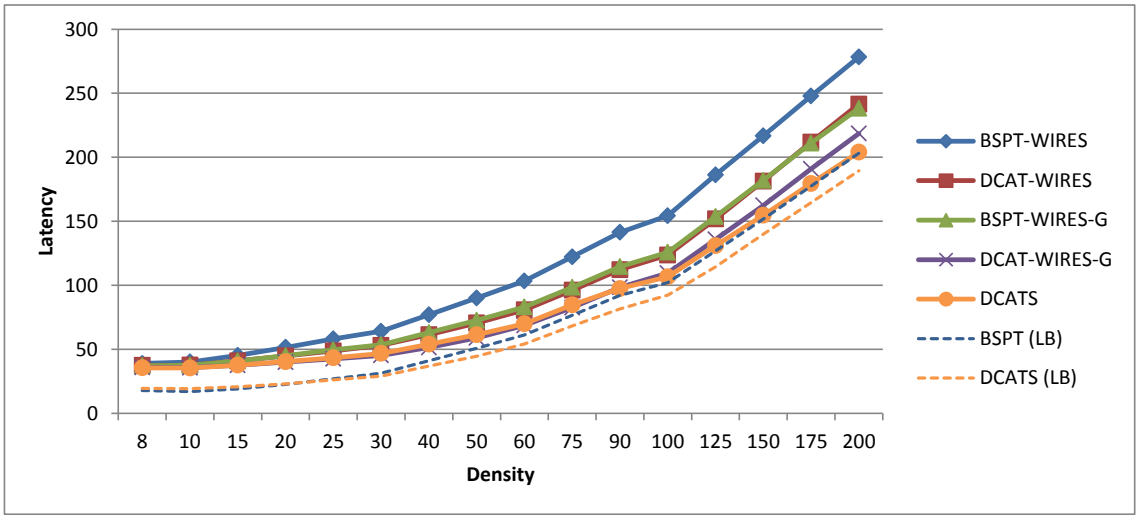

Figure 26: Average aggregation convergecast latency for a network size of 20 by 20.

BSPT-WIRES-G at almost all densities (see Table 9). This is in contrast with what we saw for small graphs where BSPT-WIRES-G was better at all densities. For large graphs, only the 3 lowest and 2 highest densities show the BSPT-WIRES-G algorithm having a lower average latency than DCAT-WIRES. This proves that starting with a good tree has an important impact on the overall schedule. As expected and if we exclude DCATS, DCAT-WIRES-G is once again the best at all densities and it is up to 15% better than BSPT-WIRES-G at medium densities, which is significant considering that they use the same scheduling algorithm.

Density	BSPT-WIRES	DCAT-WIRES	BSPT-WIRES-G	DCAT-WIRES-G	DCATS
8	38.92	37.33	37.11	35.82	**35.48**
10	40.03	37.53	37.41	35.55	**35.37**
15	45.22	40.94	40.78	**37.46**	37.49
20	51.41	44.77	44.96	**39.92**	40.3
25	58.04	48.74	49.5	**42.42**	43.31
30	63.98	53	53.47	**45.14**	46.73
40	76.97	61.35	62.93	**51.38**	53.93
50	90.08	70.47	72.58	**58.86**	61.41
60	103.21	80.77	83.03	**68.59**	69.92
75	122.12	96.05	98.34	**82.51**	84.68
90	141.39	112.18	114.52	98.38	**97.41**
100	154.34	123.64	125.76	109.49	**106.65**
125	186.22	151.71	153.59	135.75	**130.85**
150	216.69	181.28	182	162.36	**154.58**
175	247.88	211.81	210.99	190.98	**179.47**
200	278.35	241.51	238.29	218.51	**204.03**

Table 9: Average aggregation convergecast latency for a network size of 20 by 20.

For the first time, the latency of DCATS is above the lower bound of BSPT at all densities. DCAT-WIRES-G is even able to beat DCATS at densities between 15 and 75 inclusively. DCAT-WIRES-G also obtains the highest gains when compared to BSPT-WIRES, with 34.66% at a density of 50 (see Figure 27). DCATS reaches its peak at a higher density of 60 and with a smaller gain of 32.25% over BSPT-WIRES.

These results suggests that the greedy approach of selecting a parent exclusively

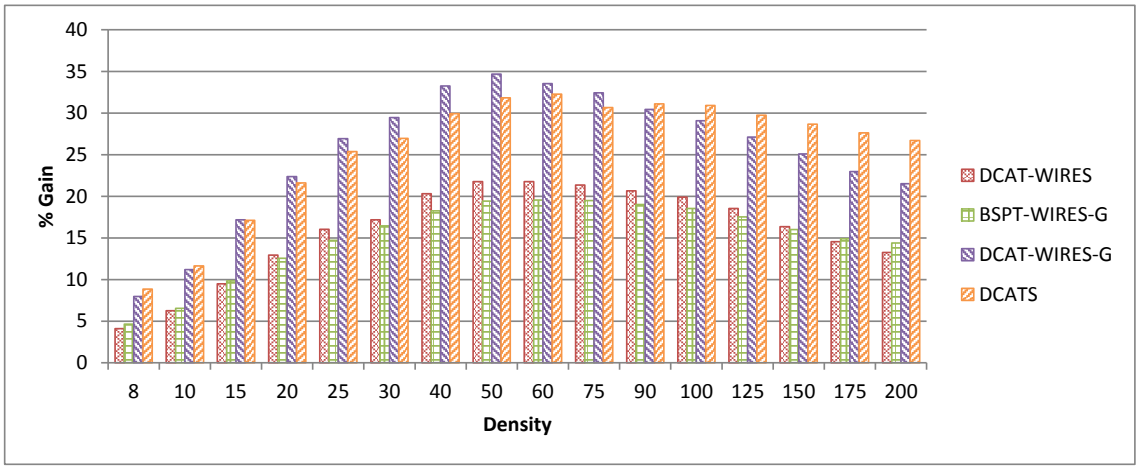

Figure 27: Average gains of using DCAT, WIRES-G or DCATS when compared to WIRES-BSPT for a network size of 20 by 20.

Density	DCAT-WIRES	BSPT-WIRES-G	DCAT-WIRES-G	DCATS
8	4.09	4.65	7.97	**8.84**
10	6.25	6.55	11.19	**11.64**
15	9.46	9.82	**17.16**	17.09
20	12.92	12.55	**22.35**	21.61
25	16.02	14.71	**26.91**	25.38
30	17.16	16.43	**29.45**	26.96
40	20.29	18.24	**33.25**	29.93
50	21.77	19.43	**34.66**	31.83
60	21.74	19.55	**33.54**	32.25
75	21.35	19.47	**32.44**	30.66
90	20.66	19.00	30.42	**31.11**
100	19.89	18.52	29.06	**30.90**
125	18.53	17.52	27.10	**29.73**
150	16.34	16.01	25.07	**28.66**
175	14.55	14.88	22.95	**27.60**
200	13.24	14.39	21.50	**26.70**

Table 10: Average latency gains of using DCAT, WIRES-G or DCATS when compared to BSPT-WIRES for a network size of 20 by 20.

on its number of neighbors might have some drawbacks. Indeed, it seems like the more balanced approach of first considering the parents on the shortest paths to the sink is better in some cases. Still, DCATS is able to outperform the other algorithms at low and very high densities and it beats BSPT-WIRES in every single test with a minimum gain of 2.27%. In comparison, there were 2 tests in which DCAT-WIRES-G produced a schedule that had the same latency as the one produced by BSPT-WIRES. We also noted that the standard deviation of DCATS results was lower than those of DCAT-WIRES-G (4.48 vs 5.35). This suggests that DCATS might have a more predictable performance.

4.5 Performance Analysis

In this section, we analyze the strengths and weaknesses of our heuristic algorithms DCAT, WIRES-G and DCATS.

4.5.1 DCAT Performance

We look at the simulation results presented in Section 4.3 to get a better understanding of why the DCAT algorithm is better than BSPT in almost all the test cases. We also investigate the rare case where BSPT outperforms DCAT by 14.95% and we give some ideas that could improve the performance of DCAT in this case.

We start by looking at the heuristic used in DCAT. The heuristic is quite simple and only looks at the degree in the graph when selecting a parent. The potential parent with the lowest degree is always selected, no matter what is its current degree in the aggregation tree. Figure 28 gives some insight as to why this simple heuristic beats a more complex algorithm like BSPT. The figure shows the relation between the number of neighbors in the graph and the average number of children in the

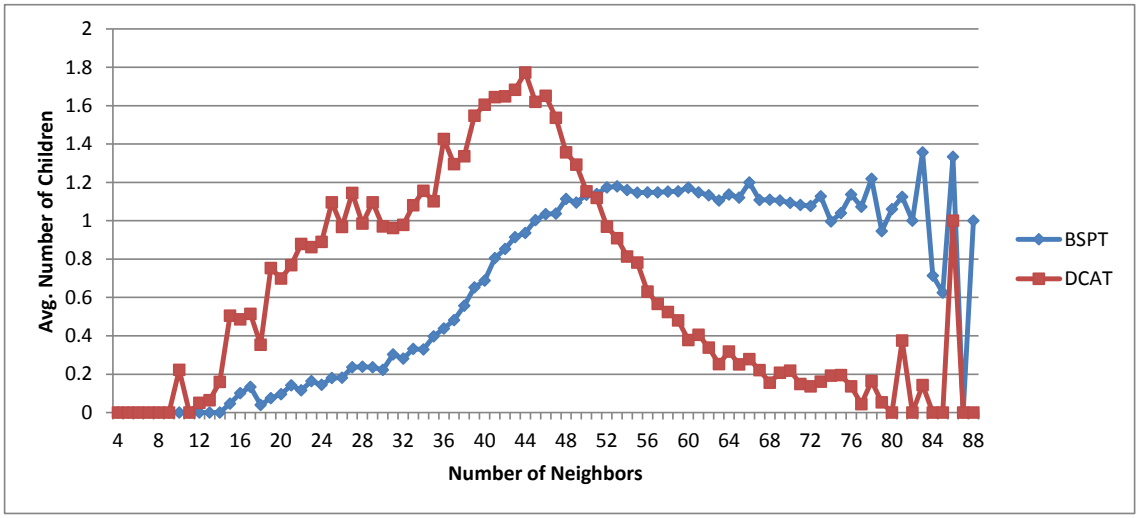

Figure 28: Relationship between the degree in the graph and the average number of children in the aggregation tree for a density of 30.

aggregation tree. We can see that BSPT assigns a lot more children to high-degree nodes than DCAT, whereas DCAT tends to assign more children to nodes that have a degree below the network density. Having less high-degree nodes as parents in the tree gives the scheduling algorithm a chance to schedule more nodes at the same time slot, which helps reduce the overall latency of the schedule.

The pattern is similar at higher densities, as shown in Figure 29 for a density of 100. Note that the BSPT algorithm tends to distribute the children more evenly between the parents. On the other hand, DCAT sometimes assigns a high number of nodes per parent (see Figures 30 and 31). This is particularly evident at a density of 100 where a little more than 5000 nodes end up with a number of children higher or equal to 15. This is compared to the fact that BSPT has never assigned more than 14 children to the same node for the same tests.

Simulation results have shown that assigning more children to lower-degree nodes is better in almost all cases, even if some nodes end up with a high number of children. However, the fact that some nodes have a high number of children might give us an

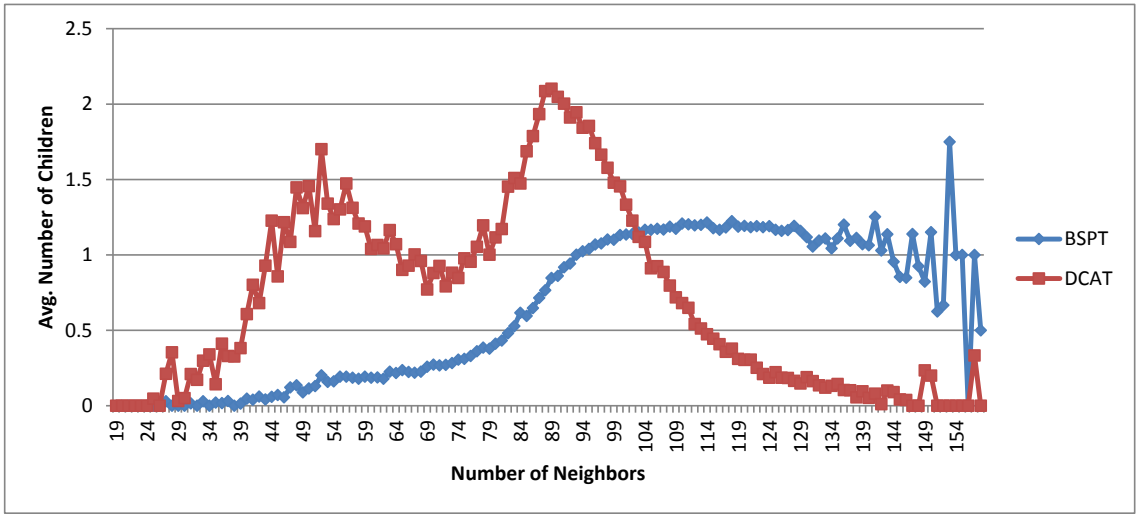

Figure 29: Relationship between the degree in the graph and the average number of children in the aggregation tree for a density of 100.

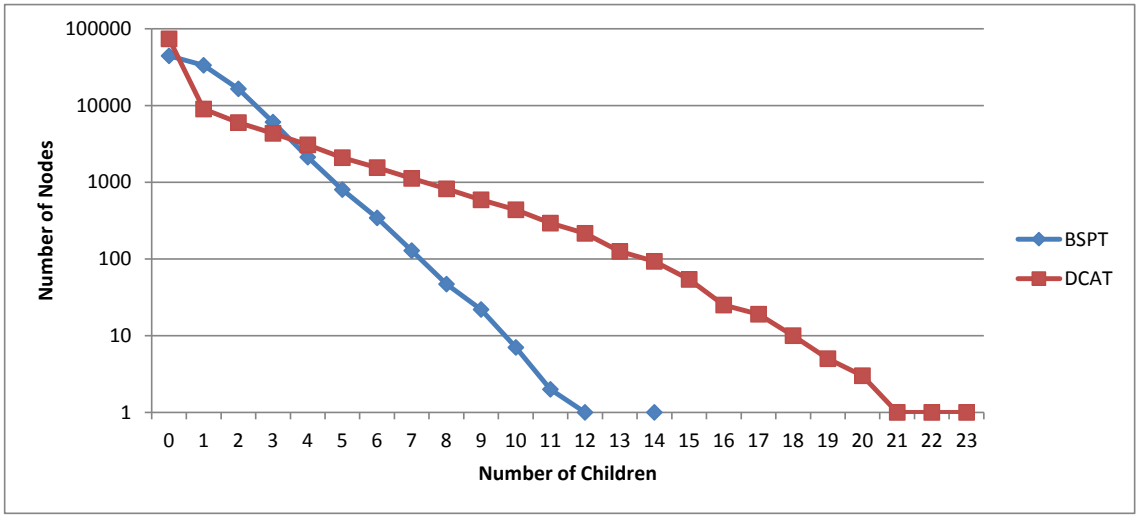

Figure 30: Number of nodes that have a certain number of children in the aggregation tree (density=30).

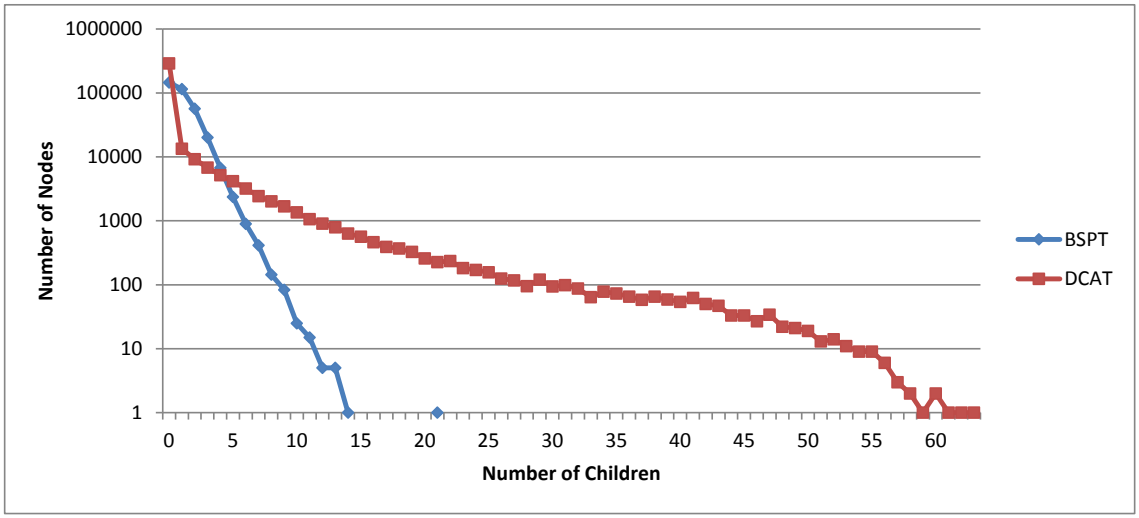

Figure 31: Number of nodes that have a certain number of children in the aggregation tree (density=100).

idea as to why there are some rare cases where DCAT does not perform well. For example, if the degree between the potential parents is very close, DCAT will always select the lowest degree node, even if it already has a high number of children. When one node has a very high degree in the aggregation tree, it becomes a bottleneck for any scheduling algorithm that doesn't alter the tree.

We tried several approaches to improve the heuristic and get a better aggregation tree. One idea was to handle cases where more than one potential parent have the same degree, and to resolve ties by looking at the current number of assigned children in the tree. This new heuristic did improve the results in some cases, but it was worse in other cases and the overall performance was similar. Another way to resolve ties was to use the parent that is physically the closest to the child, but the results were mixed in that case too. Finally, instead of systematically taking the parent with the lowest degree, we tried to add some randomness by calculating a random number and selecting the parent based on this number. The calculation was done in such a way that the potential parents with the lowest degrees were a lot more likely to be

selected, and the ones with the highest degrees had very little chance. The results of this approach were generally worse than DCAT, and by tweaking the calculation could not make it better than DCAT. In the end, we didn't find a better heuristic that was significantly better than the simple one used in DCAT.

4.5.2 WIRES-G and DCATS Performance

We look at the simulation results presented in Section 4.4 and we analyze the aggregation trees generated by the WIRES-G and DCATS algorithms. We determine what makes them perform well in our experiments and we investigate their flaws to see where they could be improved.

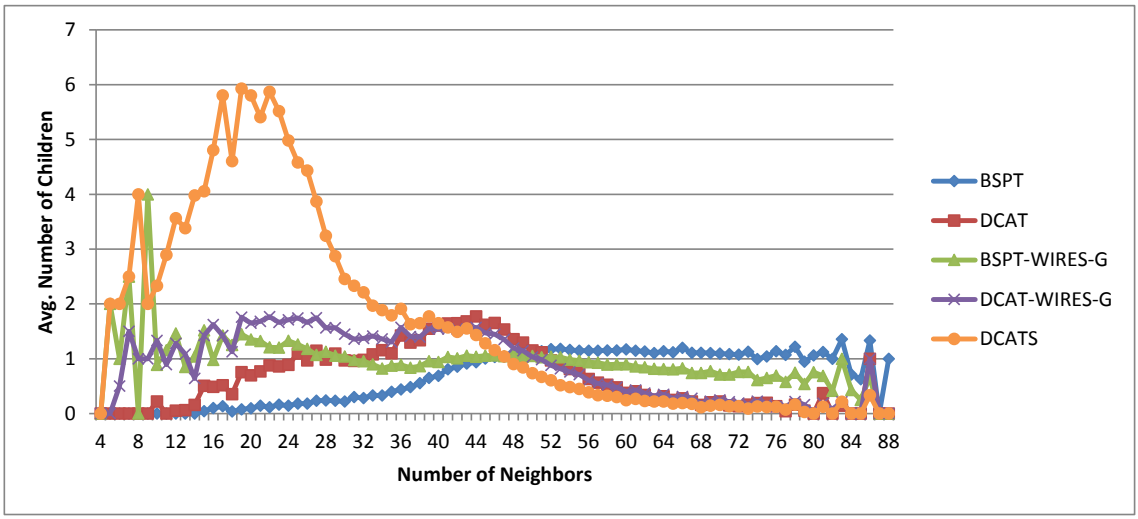

Figure 32: Relationship between the degree in the graph and the average number of children in the aggregation tree for a density of 30.

We start by looking at the relationship between the degree in the graph and the average number of children in the aggregation tree. Figure 32 shows this relationship for the medium-sized random graphs at a density of 30. As expected, we can see that the DCAT-based algorithms assign a very low number of children to high-degree nodes. The BSPT-WIRES-G combination has the best distribution of children among

all the algorithms, although it doesn't translate into the best overall performance. The DCATS algorithm assigns the highest number of children to low-degree nodes, and the lowest number of children to high-degree nodes. This is one reason why it performs well in our experiments. Figure 33 shows almost exactly the same pattern at a higher density of 100.

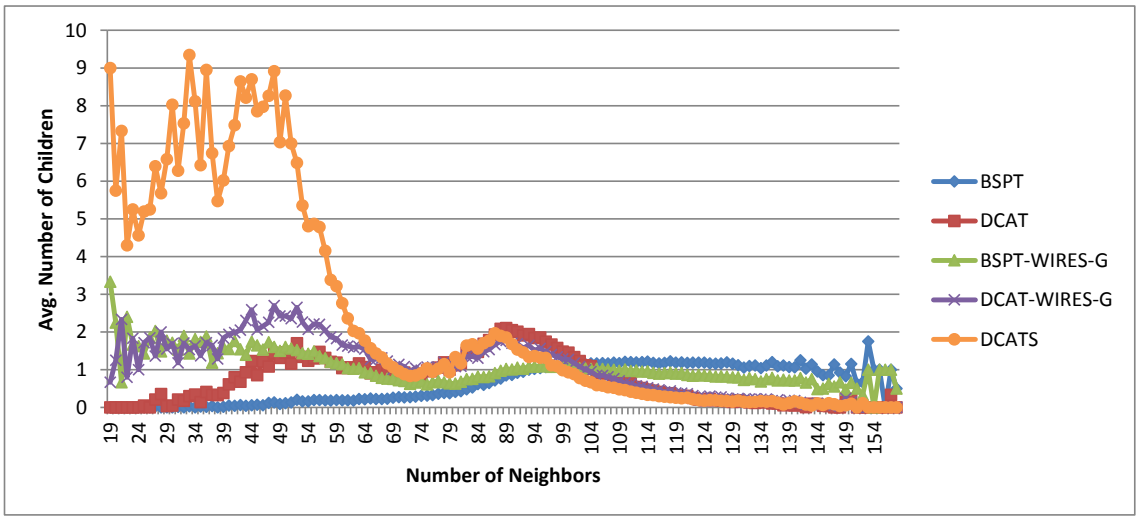

Figure 33: Relationship between the degree in the graph and the average number of children in the aggregation tree for a density of 100.

Another interesting property to look at in the aggregation trees is the location of the nodes with the highest number of children. To measure this property for a given density, we take all the nodes in all the aggregation trees and we take the thousand nodes with the highest number of children in the tree. The results for medium-sized graphs and for a density of 30 are presented in Figure 34. We can see that the sink (the node at distance 0) has always a very high number of children with all algorithms except for DCATS. This can be easily explained by the fact that we start with a shortest path tree. Therefore, all the nodes that are one hop away from the sink will be children of the sink, and the degree of the sink becomes a lower bound for the scheduling algorithm. This is particularly bad for small networks where a high

sink degree has a bigger impact on the overall latency.

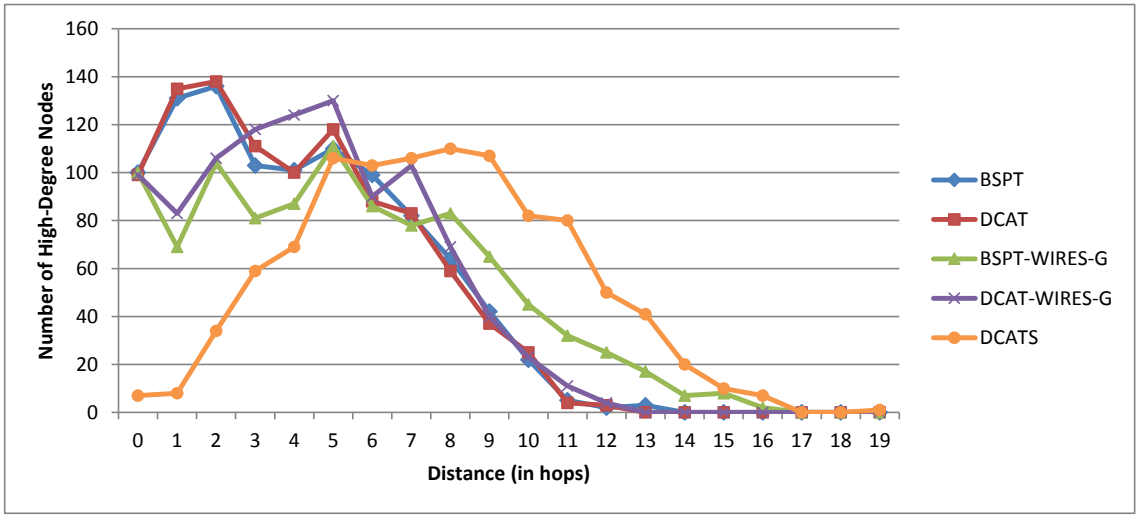

Figure 34: Location of the high-degree nodes in the aggregation tree (density=30).

DCATS doesn't have this problem because it alters the original tree to schedule as many nodes as possible at the same time. In most cases, the sink ends up with a low number of children and it is even more obvious at high densities (see Figure 35). At high densities, DCATS almost never assigns a high number of children to the sink.

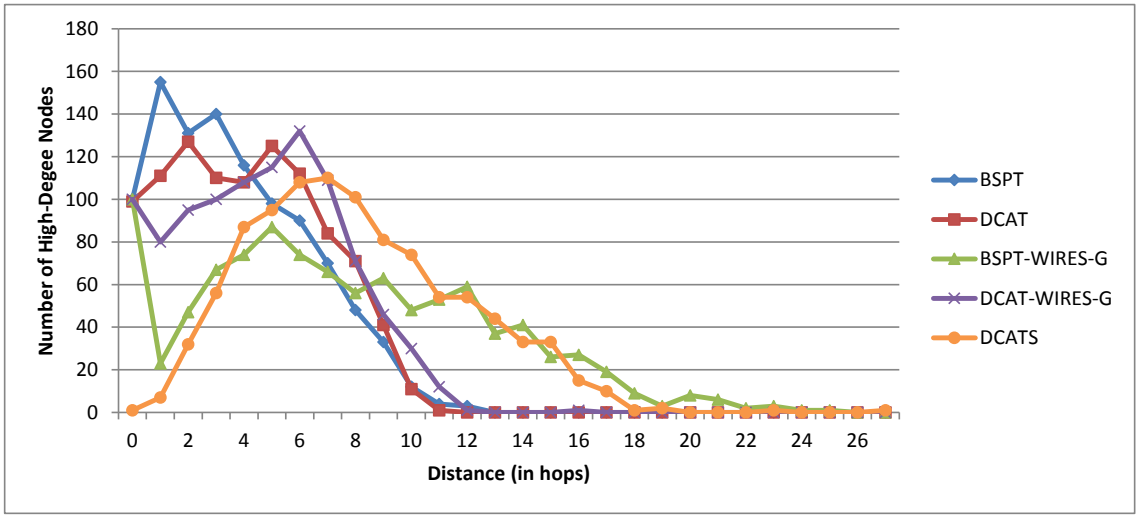

Figure 35: Location of the high-degree nodes in the aggregation tree (density=100).

We would think that the WIRES-G algorithm should also have a similar behavior

regarding the number of children assigned to the sink, as it uses the same GREEDY-SCHEDULING procedure. The problem with WIRES-G is that it first schedules as many nodes as possible without altering the tree, and then it calls the GREEDY-SCHEDULING procedure to schedule more nodes. It works well at the bottom of the tree, but once we reach the sink, the additional step is unable to schedule more nodes. Indeed, as soon as one node is scheduled to transmit to the sink, no other neighbor of the sink can transmit at the same time without causing a conflict. This explains why DCATS perform so well compared to DCAT-WIRES-G on small graphs with high densities

We observed that having high-degree nodes close to the top of the tree generally leads to higher latency. This can be explained by the fact that the possibility of parallelism is reduced as we get closer to the sink, because there are fewer nodes to choose from. Having high-degree nodes at the bottom of the tree is also generally bad, because the information cannot go up the tree until all children have transmitted. This is where DCATS is bad compared to the other algorithms and might explain why it is outperformed by DCAT-WIRES-G on large graphs with medium densities. A hybrid approach that combines the benefits of both algorithms could possibly be designed to improve the results on larger graphs. For example, the WIRES-G algorithm could be used at the lower levels in the tree and the DCATS algorithm could be used at higher levels. Such an approach would have the potential of avoiding high-degree nodes at both ends of the tree.

Chapter 5

Conclusion and Future Work

In this thesis, we looked at the problem of data gathering in wireless sensor networks. We discussed the importance of in-network data aggregation to save energy and reduce the latency of the data gathering operation. We specifically looked at applications where all the information can be aggregated into a single message using an aggregation function. We studied the problem of finding a minimum latency aggregation tree and transmission schedule. This problem is referred to as MLAS (Minimum Latency Aggregation Scheduling) in the literature and has been proven to be NP-Complete even for unit disk graphs [8]. We presented a new simpler proof of the NP-Completeness of the MLAS Problem for arbitrary networks by reducing from the well known 3-SAT problem. Using the same technique and gadget, we proposed a reduction from a restricted version of the planar 3-SAT problem to show that the problem is also NP-Complete for unit disk graphs.

We gave algorithms that build optimal aggregation trees and schedules for three specific topologies: grids, tori and trees. For regular unit interval graphs, we provided an algorithm that builds a schedule which is guaranteed to have a latency that is within one time slot of the optimal latency. Finally, for unit interval graphs we gave

a 2-approximation algorithm to solve the same problem. The MLAS problem has not been proven to be NP-Complete for unit interval graphs, so the problem of finding a polynomial-time algorithm for building an optimal tree and schedule for unit interval graphs remains open. Finding an algorithm with a constant approximation ratio for unit disk graphs is also an open problem, as the best currently known approximation ratio is $\triangle - 1$ [8].

For arbitrary graphs, we gave a new algorithm called DCAT for building an aggregation tree. Simulation results show that DCAT outperforms the previous best tree-building algorithm in terms of latency of the schedule by up to 22%, and that it is better at all the tested densities. We added a greedy step to a known scheduling algorithm called WIRES, and we called the modified algorithm WIRES-G. In our simulations, WIRES-G is better than WIRES by up to 15%, and it is also better at all densities. Furthermore, we proposed DCATS, a new algorithm that combines DCAT with a modified version of the greedy scheduling introduced in WIRES-G. This new algorithm ended up being the best in almost all tests, being only outperformed on large graphs by the combination of DCAT and WIRES-G and at medium densities. The performance of DCATS is particularly impressive on small graphs where its greedy approach allows it to be really efficient at high densities, with latency gains of up to 38.51%.

It was shown by our analysis that the approach used in DCATS is better suited to schedule the higher levels in the aggregation tree, and that the approach used in WIRES-G is better at the lower levels. In the future, a new algorithm could combine both approaches by using WIRES-G at the bottom of the tree and switching to DCATS when getting closer to the top. Such an algorithm would have the potential to significantly reduce the overall latency of the schedule on large graphs. Another possible approach would be to try to build the tree and the schedule in one single

phase, instead of the two-phase algorithms that currently exist. Our DCATS algorithm is already a step in this direction, as it performs significant modifications of the original tree. Finally, a CDS-based approach where the CDS would contain as many low-degree nodes as possible could also lead to interesting results. Indeed, previous CDS-based approaches failed to achieve good results mainly because they tend to assign many children to high-degree nodes. An approach that would instead assign many children to low-degree nodes, similar to what we do in DCAT but without the shortest path constraint, could probably perform well in practice.

Bibliography

[1] TinyOS home page. `http://www.tinyos.net/`.

[2] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, 2002.

[3] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

[4] Jamal N. Al-Karaki and Ahmed E. Kamal. Routing techniques in wireless sensor networks: a survey. *IEEE Wireless Communications*, 11(6):6–28, 2004.

[5] Valliappan Annamalai, Sandeep KS Gupta, and Loren Schwiebert. On tree-based convergecasting in wireless sensor networks. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC'03)*, volume 3, pages 1942–1947. IEEE, 2003.

[6] John Augustine, Qi Han, Philip Loden, Sachin Lodha, and Sasanka Roy. Tight analysis of shortest path convergecast in wireless sensor networks. *International Journal of Foundations of Computer Science*, 24(01):31–50, 2013.

[7] Supriyo Chatterjea and Paul Havinga. A dynamic data aggregation scheme for wireless sensor networks. In *ProRISC 2003, 14th Workshop on Circuits, Systems and Signal Processing*, 2003.

[8] Xujin Chen, Xiaodong Hu, and Jianming Zhu. Minimum data aggregation time problem in wireless sensor networks. In *Proceedings of the First International Conference on Mobile Ad-hoc and Sensor Networks*, MSN'05, pages 133–142, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] Imrich Chlamtac and Shay Kutten. On broadcasting in radio networks–problem analysis and protocol design. *IEEE Transactions on Communications*, 33(12):1240–1246, 1985.

[10] Hongsik Choi, Ju Wang, and E.A. Hughes. Scheduling on sensor hybrid network. In *Proceedings of the 14th International Conference on Computer Communications and Networks (ICCCN)*, pages 503–508, 2005.

[11] Ilker Demirkol, Cem Ersoy, and Fatih Alagoz. MAC protocols for wireless sensor networks: a survey. *IEEE Communications Magazine*, 44(4):115–121, 2006.

[12] Sinem Coleri Ergen and Pravin Varaiya. TDMA scheduling algorithms for wireless sensor networks. *Wireless Networks*, 16(4):985–997, 2010.

[13] Shashidhar Gandham, Ying Zhang, and Qingfeng Huang. Distributed minimal time convergecast scheduling in wireless sensor networks. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, pages 50–50. IEEE, 2006.

[14] Shashidhar Gandham, Ying Zhang, and Qingfeng Huang. Distributed time-optimal scheduling for convergecast in wireless sensor networks. *Computer Networks*, 52(3):610–629, 2008.

[15] Nicholas J. A. Harvey, Richard E. Ladner, László Lovász, and Tami Tamir. Semi-matchings for bipartite graphs and load balancing. *Journal of Algorithms*, 59(1):53–78, 2006.

[16] Tian He, Brian M. Blum, John A. Stankovic, and Tarek Abdelzaher. AIDA: adaptive application-independent data aggregation in wireless sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(2):426–457, 2004.

[17] Sandra M. Hedetniemi, Stephen T. Hedetniemi, and Arthur L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18(4):319–349, 1988.

[18] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd Annual Hawaii International Conference*, page 10. IEEE, 2000.

[19] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 1(4):660–670, 2002.

[20] Wendi Rabiner Heinzelman, Joanna Kulik, and Hari Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 174–185. ACM, 1999.

[21] SC-H. Huang, Peng-Jun Wan, Chinh T. Vu, Yingshu Li, and Frances Yao. Nearly constant approximation for data aggregation scheduling in wireless sensor networks. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM)*, pages 366–372. IEEE, 2007.

[22] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, pages 56–67. ACM, 2000.

[23] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, 2003.

[24] Alex Kesselman and Dariusz R. Kowalski. Fast distributed algorithm for convergecast in ad hoc geometric radio networks. *Journal of Parallel and Distributed Computing*, 66(4):578–585, 2006.

[25] Bhaskar Krishnamachari, Deborah Estrin, and Stephen Wicker. The impact of data aggregation in wireless sensor networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 575–578. IEEE, 2002.

[26] Joanna Kulik, Wendi Heinzelman, and Hari Balakrishnan. Negotiation-based protocols for disseminating information in wireless sensor networks. *Wireless Networks*, 8(2/3):169–185, 2002.

[27] Sandeep Kulkarni. TDMA service for sensor networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 604–609, 2004.

[28] David Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982.

[29] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review*, 36(SI):131–146, 2002.

[30] Baljeet Malhotra, Ioanis Nikolaidis, and Mario A. Nascimento. Aggregation convergecast scheduling in wireless sensor networks. *Wireless Networks*, 17(2):319–335, 2011.

[31] Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive precision setting for cached approximate values. *ACM SIGMOD Record*, 30(2):355–366, 2001.

[32] Ramesh Rajagopalan and Pramod K. Varshney. Data aggregation techniques in sensor networks: A survey. *IEEE Communications Surveys & Tutorials*, 8(4):48–63, 2006.

[33] Eugene Shih, Seong-Hwan Cho, Nathan Ickes, Rex Min, Amit Sinha, Alice Wang, and Anantha Chandrakasan. Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 272–287. ACM, 2001.

[34] Peter J. Slater, Ernest J. Cockayne, and Sandra T. Hedetniemi. Information dissemination in trees. *SIAM Journal on Computing*, 10(4):692–701, 1981.

[35] Sarma Upadhyayula, Valliappan Annamalai, and Sandeep KS Gupta. A low-latency and energy-efficient algorithm for convergecast in wireless sensor networks. In *IEEE Global Telecommunications Conference (GLOBECOM'03)*, volume 6, pages 3525–3530. IEEE, 2003.

[36] Peng-Jun Wan, Khaled M. Alzoubi, and Ophir Frieder. Distributed construction of connected dominating set in wireless ad hoc networks. In *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1597–1604. IEEE, 2002.

[37] Peng-Jun Wan, Scott C.-H. Huang, Lixin Wang, Zhiyuan Wan, and Xiaohua Jia. Minimum-latency aggregation scheduling in multihop wireless networks. In *Proceedings of the 10th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 185–194. ACM, 2009.

[38] Minji Wu, J. Jianliang Xu, X. Xueyan Tang, and W.-C. Wang-Chien Lee. Top-k monitoring in wireless sensor networks. *IEEE Transactions on Knowledge and Data Engineering*, 19(7):962–976, 2007.

[39] XiaoHua Xu, ShiGuang Wang, XuFei Mao, ShaoJie Tang, and Xiang Yang Li. An improved approximation algorithm for data aggregation in multi-hop wireless sensor networks. In *Proceedings of the 2nd ACM International Workshop on Foundations of Wireless Ad Hoc and Sensor Networking and Computing*, FOWANC '09, pages 47–56, New York, NY, USA, 2009. ACM.

[40] Bo Yu, Jianzhong Li, and Yingshu Li. Distributed data aggregation scheduling in wireless sensor networks. In *Proceedings of the 28th Conference on Computer Communications (INFOCOM)*, pages 2159–2167. IEEE, 2009.

[41] Jianming Zhu and Xiaodong Hu. Improved algorithm for minimum data aggregation time problem in wireless sensor networks. *Journal of Systems Science and Complexity*, 21(4):626–636, 2008.