A MODEL DRIVEN DATA GATHERING ALGORITHM FOR WIRELESS SENSOR NETWORKS

by

DHINU JOHNSON KUNNAMKUMARATH

B.Tech., Cochin University of Science and Technology,
India, 2003

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department Of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2008

Approved by:

Major Professor
Dr. Gurdip Singh

Abstract

Wireless sensor networks are characterized by severe energy constraints, one to many flows and low rate redundant data. Most of the routing algorithms for traditional networks are address centric, and the ad hoc nature of wireless sensor network makes them unsuitable for practical applications. Also the algorithms designed for mobile ad hoc networks are unsuitable for wireless sensor networks due to severe energy constraints that require nodes to perform for months with limited resources, as well as the low data rate which the constraint implies.

This thesis examines a model driven data gathering algorithm framework for wireless sensor networks. It was designed with a goal to decrease the overall cost in transmission by lowering the number of messages transmitted in the network. A combination of data-centric and address-centric approaches was used as guidelines during the design process. A shortest path heuristic where intermediate nodes forward interest messages whenever it is of lower cost is one of the heuristics used. Another heuristic used is the greedy incremental approach to build a lower cost tree from a graph with various producers and consumers. A cost division heuristic is used to divide cost of shared path into distinct paths as the path forks in a tree.

This thesis analyzes the effects of these heuristics on the performance of the algorithm and how it lowers the overall cost with the addition of each heuristic.

Table of Contents

List of Figures

List of Tables

List of Code

Acknowledgments

Here I express my sincere gratitude to my major adviser Dr Gurdip Singh. I also thank Dr Howell and Dr Banerjee for serving in my committee; as well as Dr Andresen who substituted for Dr Banerjee.

I also thank my friends from Kansas State. They have made my stay here in Manhattan delightful. In particular, I have to express my gratitude to Stroade family who let me stay with them this last year.

Last but not the least, I would thank my family who supported me through and through. And to the One above all who has watched over and guided me.

Chapter 1

Introduction

Since the invention of the first transistor at Bell Labs in 1947 by William Shockley, John Bardeen and Walter Brattain, developments in modern electronics have progressed in a rapid rate. Gordon Moore observed in 1965 that computing power doubled per unit cost every two years. Results of this evolution in electronics are seen today in the plethora of cost effective electronic devices equipped with wireless communication.

It is interesting to observe that the original Internet Protocol suite was inspired by the early packet radio systems of 1970s. Packet radio networks were one of the earliest wireless ad hoc networks. Though most projects at that time were funded by DARPA for military utility, the evolution of this field has become a significant influence in our everyday life.

Ad hoc network is a term used today to characterize networks that lack fixed infrastructure and pre-configuration, and have the capability to dynamically organize and communicate. The performance of conventional networks depends on continuous connectivity, bandwidth, reliable power supply and static configuration and topology. An ad hoc network does not rely on these supporting infrastructures. As suggested by Murphy in his paper on formal reasoning for mobile communications[1], an ad hoc network can be visualized as a continuously changing graph. Communication depends not only on the distance between nodes, but also on the willingness of individual nodes to collaborate and form a cohesive transitory community. A wireless ad hoc network is formed on the fly, and changes as the nodes enter, relocate or leave the network.

Mobile ad hoc network(MANET) is a type of self configuring ad hoc network consisting of mobile routers and associated hosts. It became popular in 1990s with the widespread usage of the IEEE 802.11 wireless communication standard and laptops. According to Aggelou[2], "A mobile Ad hoc network is a system of mobile wireless nodes that can freely and dynamically self-organize in arbitrary and temporary network topologies without the need of a wired backbone or a centralized administration."

A Wireless Sensor Network(WSN) is also a type of ad hoc network. The network consists of a large number of immobile nodes spanned in a large geographical area. These networks are configured to perform a certain application specific task. Each node is equipped with a sensor or/and an actuator. A sensor node has wireless communication capability and some level of intelligence for signal processing and networking of the data.[3]

1.1 Wireless Sensor Networks

Wireless Sensor Networks are identified as one of the most important technologies of this century.[4] Advances in wireless communication and modern electronics have led to the development of low-cost, low-power, multi-functional sensor nodes which are small sized and can communicate short distances. Sensor nodes consist of components capable of sensing, data processing and communication.[5]

A WSN has a large density of sensor nodes deployed in a geographical area. Each of these node is equipped with a sensor that can sense the environment. But they can also be equipped with an actuator. An actuator is a component that can be used to interact with environment using a mechanical part based on inputs. The distribution of the nodes thus forms a radio network inherently ad hoc. Hence WSN can be visualized as a graph that connects nodes in a momentary and decentralized fashion.

The characteristics that distinguish a wireless sensor network from an ad hoc network were listed in Akayildiz et al.[5] They are as follows.

- Node density in a WSN has a higher magnitude than an ad hoc network.

- Number of nodes in a WSN is also of a magnitude several orders higher.
- Sensor nodes can fail often.
- Topology of WSN changes frequently due to node and link failures rather than redeployment.
- Sensor nodes mainly use broadcast communication, where as other ad hoc networks like MANET use point to point communication.
- Sensor nodes have limited resources such as power, memory and computational capacity.

The impact of these characteristics can be observed in WSN. There will be redundant data sources, since the data collected by the sensors are from a common phenomenon. The data are transmitted to a sink from the sensor nodes, thus making the path of data transmission to data recipient a reverse multi-cast tree in most applications. Since most of the sensors are immobile and required to operate for months without any intervention, there is a severe constraint on energy consumption of sensor nodes. A sensor node consumes power mainly during sensing, communication and data processing. Energy consumption can be minimized by reducing the frequency of activities in WSN, thus suggesting that high data rate transfers are not feasible. Besides in traditional networks, data are transmitted from source to sink via the shortest route possible. This address centric approach is not suitable for WSN applications as it can not eliminate the data redundancy occurring in most of the cases. Because of these factors, most algorithms and protocols developed for MANETs are unsuitable for wireless sensor networks.

1.2 Data aggregation

In order to conserve energy, a strategy to avoid redundant data and to aggregate data en route must be used. This data centric approach may be better suited for wireless sensor

network applications. In this approach, contents of data packets en route are examined by nodes on the path to perform some consolidation or elimination process to avoid redundancy due to data originating from different sources. To understand the practicability of usage of data aggregation, the context in which such techniques may be used needs to be analyzed. The type and dynamics of the data flowing in the wireless sensor network from the sources may be considered[6]:

1. There is no data redundancy as all sources send completely different data.

2. There is complete data redundancy as all sources send identical information.

3. There is an intermediate level of redundancy in data transmitted by sources and data is non-deterministic in nature.

When there is no redundancy, data aggregation is not possible. Hence, in this case a data centric as well as an address centric algorithm can preform with same number of data transmission in a sensor network. But when there is complete redundancy in data transmitted by sources, an address centric approach can be modified to perform better than an data centric approach. This can be observed when an address centric algorithm is modified to request data only from one of the redundant data sources. In the last case, the sink can not request some sources to merely shut down, due to the non deterministic nature of data. So address centric algorithms can not be modified to work better than a data centric approach. This case of wireless sensor networks is the subject of interest for rest of the discussion contained here.

Data aggregation may be implemented in several ways, but duplicate suppression is the simplest data aggregation function and is implemented in most wireless networks. As in Krishnamachari et al[6], for the modeling purposes the aggregation function is assumed to transmit a single aggregate packet when it receives multiple input packets.

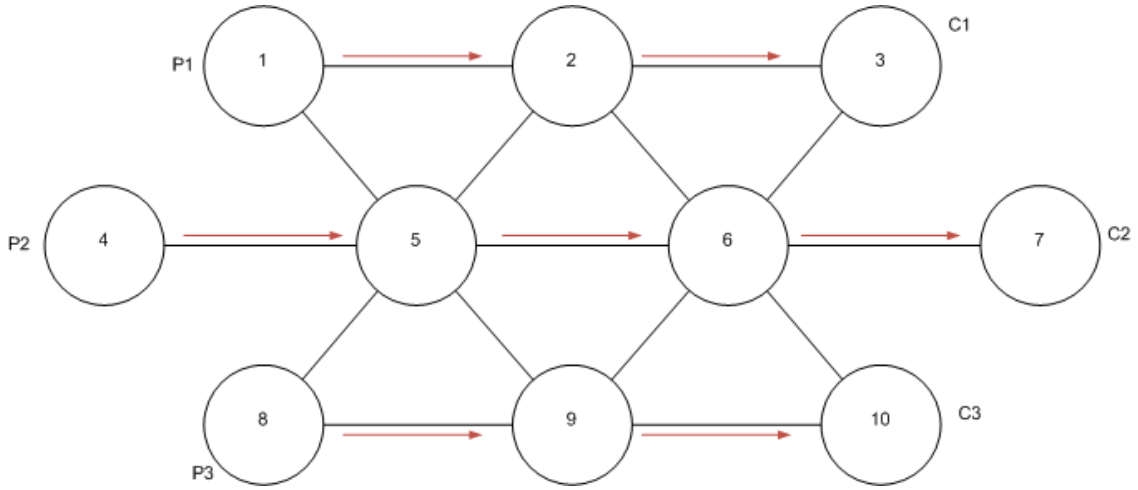


Figure 1.1: *A shortest path graph*

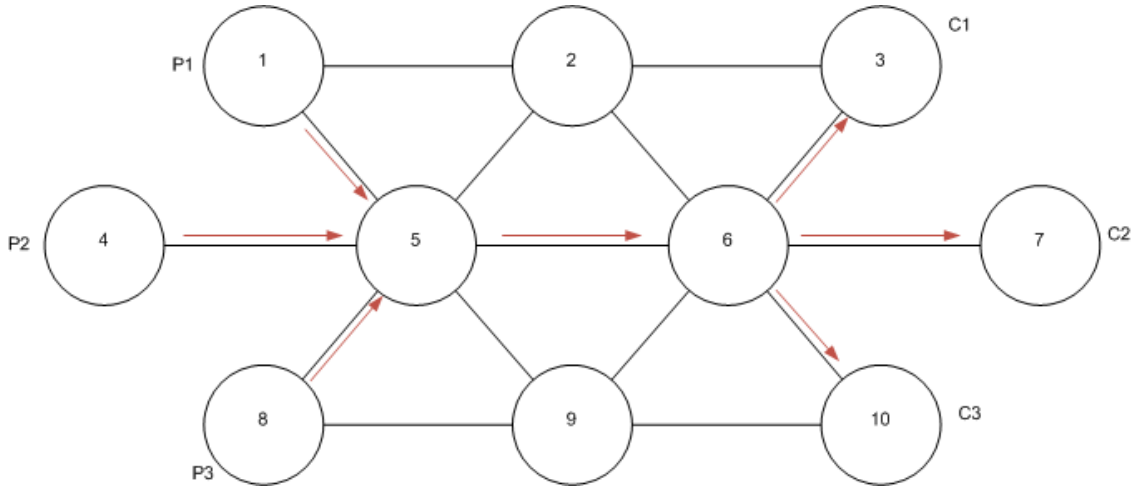


Figure 1.2: *An optimal path graph*

1.2.1 Optimal aggregation

In a network graph $G = (V, E)$ where V is the set of nodes and E is the set of edges that connect nodes which can communicate directly, let $S_1, S_2..., S_k \in S$ be data sources and D be a sink node. For optimal aggregation a minimal cost tree connecting nodes in S and node D with minimal number of edges should be found. This is the Steiner Tree problem[7] which is an NP-hard variant of the minimum spanning tree problem. At this point it is safe to conclude that, the optimal number of transmissions in a data centric algorithm, where

there is only one sink is equal to the number of edges in the minimum Steiner tree of the network which contains node set $S \cup D$.

1.2.2 Suboptimal aggregation

Now that it is clear that an optimal aggregation can not be obtained, some heuristics may be used to obtain suboptimal solutions to the data aggregation problem in the wireless sensor network. A couple of heuristics are described here.

1 Shortest Path Tree: Here the source sends data to the sink node along the shortest path between them. Wherever the paths overlap, they are combined to form the data aggregation tree.

2 Greedy Incremental Tree. At first the tree consists of a shortest path between a sink and its nearest source. To this tree, sequentially other nearest sources are added.

If there is more than one sink in the network, the complexity of data aggregation problem is increased. Now the optimal solution can contain either a singly connected graph or a forest of trees, where the number of trees is less than or equal to the number of sinks. Figure 1.2(a) a possible shortest path graph is shown. Figure 1.2(b) shows an optimal graph for the same topology. In these figures nodes P1, P2, P3 generate same data and nodes C1, C2, C3 consume them.

However, the nature of wireless sensor networks eliminates the possibility of finding these suboptimal data aggregation solution in a centralized manner. A distributed solution to this problem is discussed in this thesis.

1.3 Applying heuristics

A description of how I implemented the heuristics described in the previous section is given here. In addition to that, another heuristic to distribute cost is given towards the end of this section.

Applying the shortest path heuristics: One of the ways that is used in traditional networks to find shortest path is by flooding a beacon through the network which will cumulate the path information. Two key parts of the path information are cost and sequence of links that form the actual path.

One way to implement the shortest path algorithm is to let a consumer initiate the process of finding the shortest path by broadcasting a message indicating its interest in a certain data type. This message will be flooded through out network. As the message is propagated, at each intermediate node the cost information on the message is updated. In addition the information regarding the sink and the neighbor from which the least cost packet arrived is stored in the node. In the shortest path algorithm, if the actual shortest path is to be determined by consumers, then intermediate nodes will require forwarding of duplicate messages with lower costs. The producer can send data, addressing it to consumer and this scheme avoids flooding of data through the network. This is an example of traditional address-centric approach.

Since end-to-end schemes are not practicable for wireless sensor networks, only some of the techniques used in shortest path algorithm is used in the design of this algorithm. When the consumer sends out requests for data it may be propagated through the network, as the consumer does not have any information on data sources. Whether or not to propagate the lower cost duplicate messages at intermediate nodes is a design choice. Data that are transmitted by producers are not addressed to consumers, but rather they make a path through the network following the least cost request at intermediate nodes.

Translating the greedy incremental approach: In this heuristic, a shortest path needs to be found between a consumer and a producer. Then other nearest consumers and producers are attached to the graph incrementally. In other words, if there is already a path between a source and a sink, and an intermediate node on this path receives a message requesting a data which is available, then the node just need to reply with data.

Cost division approach: This is another heuristic that may be used to give preference

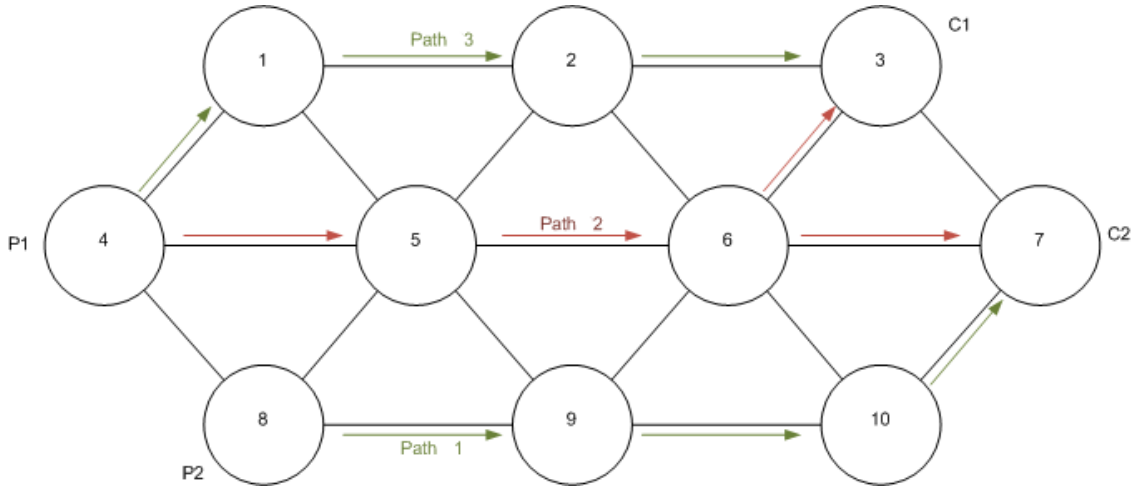


Figure 1.3: *Cost division*

to a shared path. In Figure 1.3 nodes P1 and P2 generate same type of data and nodes C1 and C2 are consumers of this data. Path 1 is a shortest path between producer P2 and consumer C2, where as path 3 is the shortest path between producer P1 and consumer C1. The overall cost of transmitting data through these two paths are higher than the shared path 2(here cost is simply the number of hops). C1 and C2 can be made to select path through node 6, if it has lower cost than paths 3 and 1 respectively. This can be made possible by cost division approach. If the cost(which is 2) of data path from node 4 to 6 is distributed into branches, then the cost(which is 2/2 + 1) of path through node 6 will be lower than that of path 1 and 3(which is 3 each).

1.3.1 Consumer Logic:

In a typical sensor network application there will be more than one node which collects data and process them. The case might be that, such consumers might require more than one data item that are produced by various nodes in the network. There are four different cases, one can consider:

1. Simple: Consumer node requires only one kind of data.

2. And Logic: Consumer node requires more than one data item, and require to collect

all of them.

3. Or Logic: Consumer requires only one of the data items of the many data it might possibly request.

4. Combination Logic: Consumer requires more than one data item and uses a combination of both logics among various data sets it requests.

This thesis considers the study of first three cases only, with the application of heuristics discussed before.

1.4 Thesis Organization

An outline of this thesis is as follows: A related work, directed diffusion paradigm and some algorithms implemented in this paradigm are discussed in Chapter 2. A brief overview of TinyOS platform and the tool TOSSIM and nesC compiler used in development and testing is also given in that chapter. Chapter 3 discusses the design of the algorithm as well as its implementation. Analysis of the algorithm is discussed in Chapter 4. Chapter 5 discusses the conclusions and some future work that may be done.

Chapter 2

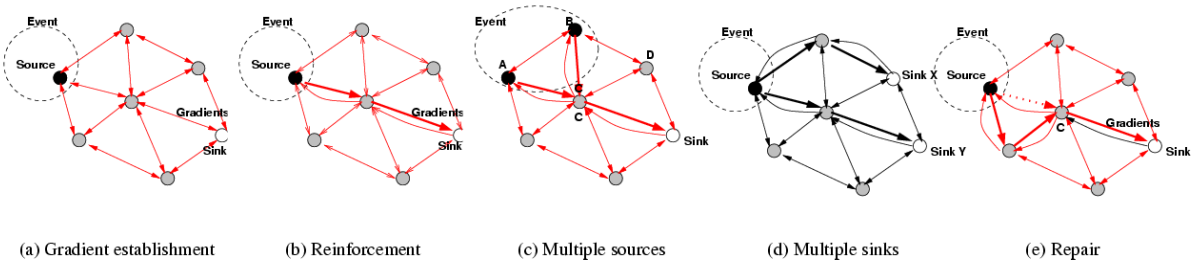
Background and Related Work

A sensor network may be viewed as a distributed event-based systems. Data is usually collected or processed information of a physical phenomenon. So data can be an event which is a short description of phenomenon sensed by the wireless sensor network. Nodes that generate or publish information are sources, while sinks are nodes that consume or subscribe this information.

A data centric approach for designing wireless sensor network applications is discussed in Intanagoniwat et al[8]. Directed diffusion is described in this paper as a paradigm for communication in distributed wireless sensor network applications. Here every sensor node data is named with an attribute-value pair. Sinks express interest in named data by generating interests. Matching data is then "drawn" towards the sink. Data can be cached, transformed or aggregated as part of in-network processing. Based on cached data, interests may be directed by intermediate nodes. Localized interactions between a node and its neighbors can be used to determine data and interest propagation as well as aggregation.

2.1 Diffusion algorithms

Data dissemination algorithms that do in-network data processing to move data from sources to sinks are called diffusion algorithms[9]. Two-phase pull diffusion was an algorithm used in initial work with directed diffusion[8]. The other algorithm discussed is one-phase pull diffusion[9].



(a) Gradient establishment (b) Reinforcement (c) Multiple sources (d) Multiple sinks (e) Repair

Figure 2.1: *Different aspects of diffusion*

2.1.1 Two-phase pull diffusion

Data used in this algorithm consists of named attribute-value pairs. Sinks use a set of attributes to identify data. This information is contained in interest message which is by default flooded through the network. Figure 2.1(a) shows a sink broadcasting an interest message and further propagation of interest messages and establishment of gradients throughout the network. Each gradient consists of a value and direction(the direction points to the neighbor from which the interest came). As a result of flooding every neighboring node pair will have gradients set up towards each other.

Once the source receives an interest, it starts sending data. The first data transmitted is exploratory, as it will be forwarded to all neighbors that have matching gradients. This results in flooding of data, as every neighboring node pair has gradients set up towards each other due to flooding of interest messages.

Figure 2.1(b) shows the establishment of reinforced gradients, once a sink receives exploratory data. The sink reinforces a neighbor, and a reinforced gradient is established to itself at the neighbor and this process repeats resulting in a graph of reinforced gradients from all sinks to sources. After this phase data messages are not marked exploratory and are transmitted only on reinforced gradients. Figure 2.1(c) shows the flow of data along the established reinforced gradients from multiple sources to a sink and Figure 2.1(d) shows the flow of data along the established reinforced gradients from one source to multiple sinks.

The gradients in the network are managed as soft-state. Interests and exploratory data

are propagated periodically to refresh gradients. Periodic transmission of reinforcement messages is required to maintain reinforced gradients. Figure 2.1(e) shows the use of negative reinforcements in this algorithm to correct routes. When a node receives data that is no longer relevant to it(for example, due to a topology change), the node no longer sends reinforcement to the node that delivered irrelevant data. This will cause gradients to time out and the data path will cease to exist. Another way to do this, would be to transmit negative reinforcement messages to the node that delivered irrelevant data.

2.1.2 One-phase pull diffusion

This is an improvement over two-phase pull diffusion, as it eliminates one of the search phases. Similar to two-phase pull, sinks generate interest messages that are diffused through the network resulting in gradient establishment. The next search phase of transmitting exploratory data is not present in one-phase pull, but instead data is transmitted only to the lowest latency gradient that corresponds to each sink. To identify various sinks, each sink needs to maintain a unique identifier, either formed from its MAC address or by using probabilistic approaches.

2.2 Observations on Diffusion Mechanism

1. There is a periodic interest propagation phase. The initial interest message can be considered exploratory as it tries to find a source for the data. The sink periodically refreshes the interest which is a soft state. Refreshing is a necessity as it is not possible to transmit interest reliably across the network. The refresh rate is a protocol design parameter. The interest may be propagated by flooding or by directional propagation based on previously cached data.

2. Each distinct interest is stored in a cache at each node. Interest aggregation is possible when a similar interest reaches the node from a different sink, as the cache does not store the sink information. An entry in the cache has various fields that corresponds

to the message fields. When a node receives an interest, first it checks if the interest exists in the cache. An interest entry is created if no matching entry exists. A received interest message is used to instantiate the parameters of the entry. This new entry will have a gradient towards the node that sent the interest. If another interest with same parameters is received by the node from a different neighbor, no new entries are made, but a gradient is set up in the direction of neighbor from which it received the latest interest. A gradient is removed from its interest entry, when it expires. An interest can be removed from the cache when all its gradients have expired.

3. Interest propagation phase is followed by data propagation phase. A source sends data as a unicast message to its neighbors that have a gradient entry in its interest cache.

4. Data will be cached at intermediate nodes. The receiving node of a data message compares parameters of the data message to its entries in interest cache. If no match is found, the data packet is silently dropped. If there is a match, data cache of the node is checked for its associated match. The data cache keeps track of recently sent data messages and it can also be used for loop prevention. A received data message will be dropped if it has a matching data cache entry. Otherwise, a new entry is made in the data cache and the data message is re-sent to node's neighbors. Duplicate suppression is the simplest form of data aggregation.

5. Once the sink receives exploratory data transmitted by a source, it sends out reinforcement messages to select the low latency path. If at a later point, another datum with lower latency than the current data path arrives at the sink, the sink may reinforce that path. With the usage of negative reinforcement the sink may cancel the existing path or by timing out of soft state(gradients) the extra data path may be removed.

6. Diffusion can be considered an on demand routing technique. It does not make an effort to construct a loop free path, but rather multiple data paths are set up. Reinforcement is used to reduce the multiplicity of data paths. And in order to avoid loops, caching

is used to keep track of recent messages.

7. A mechanism not requiring information on network topology is beneficial, because as far as a node is concerned, the message it receives comes from its neighbor. Because of these localized interactions, global topology changes are not required to be propagated through out the network. Also in-network aggregation can be used to improve energy efficiency of the algorithms.

2.3 Development Tools

2.3.1 Tiny Microthreading Operating System(TinyOS)

TinyOS consists of a tiny scheduler and a graph of components. "A component has four interrelated parts: a set of *command handlers*, a set of *event handlers*, an encapsulated fixed size *frame* and a bundle of simple *tasks*." [10] Tasks and handlers execute in the context of a frame and operate on its state.

A tinyOS application normally consists of a number of components wired together. Each component may use other components. Higher level components issue commands to lower level components and lower level components signal higher level components. The topmost level component of the application is not used by any other component, where as the other components may interact with each other using interfaces.

TinyOS uses static memory allocation, and requires every component to declare the commands it uses and events it signals, so that the memory requirements of components are known at compile time.

Commands are non-blocking requests and are required to return status to their caller. Hardware events directly or indirectly invoke event handlers. A hardware interrupt can thus trigger a series of events that do upward processing and downward processing through commands.

The majority of the work is performed by tasks. Tasks are atomic with respect to one another but can be preempted by events. They are scheduled by the tinyOS scheduler in a

FIFO manner.

2.3.2 nesC

nesC[11] is a static language and is an extension of C. It performs whole program analysis during compilation. In order to support this analysis, every component, whether a module or a configuration, has to declare all the interfaces it uses and provides. Each component is accessible only through these interfaces given in its declaration.

Configuration files are used to wire other components together, connecting interfaces used by a component with that of one provided by another component. All nesC applications have a top level configuration which connects all the components used. Modules are components that provide an implementation of commands for the interfaces it provides and events for the interfaces it uses.

Concurrency is provided through tasks and events. Events are asynchronous code, as they are triggered by hardware interrupts. The communication layer of TinyOS is not designed to be executed as asynchronous code. The nesC compiler is capable of determining race related issues during compile time.

An example of a top level component of an application that wires components together is listed in Appendix A. This is a component declaration of the simulator application which runs one of the heuristic algorithms. The code snippets with in Chapter 3 are nesC listings.

2.3.3 TOSSIM

TOSSIM[12,13] is a discrete event simulator for TinyOS applications written in nesC. The nesC compiler uses a PC option to compile TinyOS application into a TOSSIM application. TOSSIM is capable of simulating up to a thousand nodes, by replacing some low level components with hardware abstraction components.

TOSSIM compiles the nesC code, and generates a discrete event queue from TinyOS component graphs. It runs the same code that runs on the sensor network hardware. The simulator event queue generates discrete events that are equivalent to hardware interrupts

to drive the simulation. The remainder of the TinyOS code runs unchanged.

The simulator models a network as a directed graph. Each edge has a bit error probability that indicates the fidelity of transmission between nodes, which are represented by vertexes. This abstraction allows testing with various error rates. Network topology along with error rates can be provided as input to the simulation in a file.

TOSSIM also supports mechanisms for extensible radio and ADC models as well as communication services to external programs that can interact with a simulation. TinyViz, a Java based graphical user interface is one such external program.

In the next chapter, the algorithm design and implementation is discussed. The algorithm tries to adopt some of the diffusion mechanisms discussed in Section 2.2, while incorporating heuristics. The implementation of the algorithm was done for the TinyOS platform and was tested on TOSSIM.

Chapter 3

A data gathering algorithm

3.1 Algorithm Design

3.1.1 Introduction

In a wireless sensor network application, nodes that sense and generate data are called *sources*. Nodes in the wireless sensor network that require data for processing are called *sinks*. The nodes that do not perform either of these functions can be used as relays and hence are called *relays*. A sink generates a message that identifies its requirement. This message is called *interest*, and is propagated throughout the network. The source transmits data message for the *interest* messages it receives. The transmitted data causes the formation of one or more data paths in the wireless sensor network. The sink may choose to *reinforce* only one of these paths. The sink may transmit *reinforcement* message along only one data path to reinforce it.

A data message can be identified by its source and the type of data. An interest message generated by a sink will contain the data type which it requires.

3.1.2 Interest propagation

When a sink transmits the first interest message, it does not have any information on the availability of data. So the simplest choice for sink is to broadcast interest message to all its neighbors. Interest message is an exploratory message and is broadcast throughout the

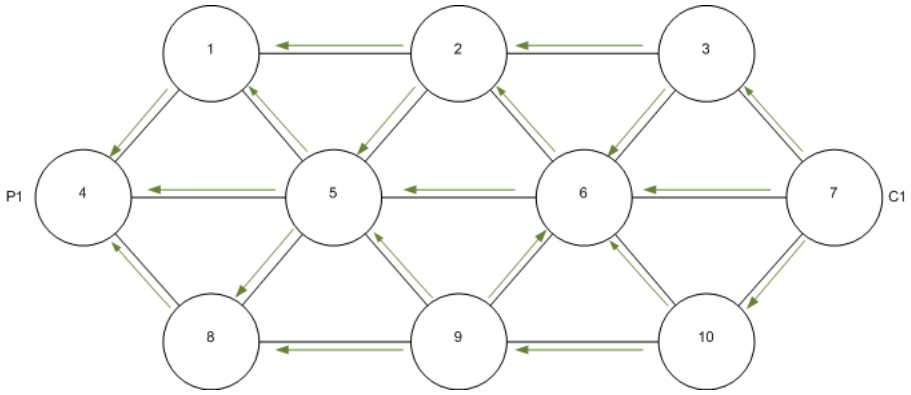


Figure 3.1: *Interest propagation*

network. Figure 3.1 shows propagation of interest message through the network.

Some of the features of the shortest path heuristic can be applied in *interest* propagation. As interest message diffuses through the network, it accumulates path information. An interest message uses a cost field to hold a simple cost indicator which is a hop count. Whenever a node receives an interest message, it will look up in its *interest table* for the received interest message. An interest table has only one entry per data type from a particular sink.

Consider the interest table of node 5 shown if Fig 3.1. If node 9 delivers the first interest message(generated by sink 7), a new entry is made in the table with its parameters instantiated from the interest message. Interest message is then relayed on by another broadcast. Later when node 6 delivers the interest message with lower cost(cost being number of hops), it is considered a duplicate as there is a matching entry in the table. There are two possible actions that node 5 can perform in response. One is to drop the packet, and the other is to forward it. If the node forwards the lower cost interest message, then algorithm is incorporating the shortest path heuristic. When node 2 delivers the interest message to node 6, it is treated as a duplicate and can be ignored, as its cost is higher than or equal to that of the existing entry.

The table is periodically refreshed to remove expired entries. The refresh period is a design issue depending on application and size of network.

3.1.3 Data propagation

As in the case of relays, a source also maintains an interest table. When a source receives an interest message, the source replies with a data message to the neighbor that transmitted the interest message for two cases:

1. Interest message is the first request from a particular sink for the data.

2. Interest message is not a first request from a sink, but has a lower cost.

In Figure 3.1, if source 4 receives the first interest message from node 8, then the source will reply with data. If node 1 delivers the interest message after node 8, then that interest message will be ignored by the source. But when node 5 delivers interest message, the source will reply with data as it is of lower cost. Thereafter, the source will send data message periodically against each distinct interest message that is reinforced. If more than one reinforced interest entry came from the same neighbor of the same data type, only one data message is transmitted to that neighbor to prevent duplicates being transmitted to the neighbor from source. As data message propagates through the network, it also accumulates a cost value, which, in this case, is number of hops the data message has traveled.

A *data table* is maintained at each relay. The table keeps track of the most recent messages. When a relay receives a data message, it checks whether data exists in the table by comparing parameters of the message with that of the table entries. Data message is duplicate if it is from the same source, has the same type and has the same sequence number (the sequence number or count is also used to identify the newness of a message). If it is duplicate data, then it is ignored. If it is new data, then the table is updated and the data message is forwarded. When data from a different source arrives at the relay, an entry is made in the table only if the cost of data message is lower than the current entries for the data type; otherwise it is ignored. Whenever an entry is made/updated in the table, it is assigned a time stamp. A table entry is removed only at its expiry.

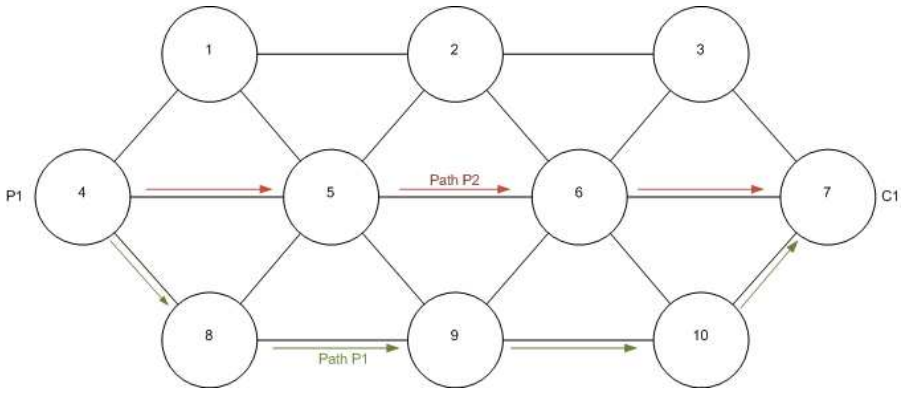


Figure 3.2: *Multiple data paths*

If a data message received at a relay is first of its type, then it is forwarded to all neighbors that have entries in its interest table. But if the data is not the first of its type, then it is forwarded to its neighbors whose entries are reinforced in its interest table. Another case that requires forwarding of a data message is when data exists in the table and a new interest message for data is received. In this case, data message will be transmitted against the interest message and the cost of the data message may be distributed over an existing path (reinforced path) as well as the would-be paths.

3.1.4 Path maintenance

When a sink receives a data message it has requested, it sends reinforcement messages to confirm the path for data delivery. If a sink confirms the first path that delivered data, then it may be a lower delay path rather than a lower cost path.

Figure 3.2 shows data paths P1 and P2. Although the path P2 has less number of hops than the path P1 between the nodes 4 and 7, the messages transmitted through the path P1 may reach node 7 earlier than the messages transmitted through the path P2. This delay can be caused by extra communication or processing load on nodes 5 and 6 in the path P2. Thus the path P1 can become a lower delay path, and the path P2 a lower cost path.

Data with lower cost can be delivered at a later point by a different neighbor than the one through which current data path is set. The sink then has two choices: either to reinforce

the new path immediately or wait until the next refresh cycle to confirm the new path. A *refresh cycle* is a period of time between transmissions of reinforcements messages at a sink. If a sink is to confirm the path immediately then it will cause duplicate data paths. This choice is avoided in the implementation.

At each relay, on reception of reinforcement message, its parameters are compared with entries in the node's interest table. If there is an interest entry for the data type from the sink which sent the reinforcement message, then the entry is updated to indicate that it is reinforced and that its expiry is extended. *Reinforcement* is forwarded to the node that relayed the least cost data for the reinforced data type.

On receiving *reinforcement*, the source updates its interest table similar to the relays, but does not forward *reinforcement*. It periodically sends out *data* to neighbors that delivered *reinforcements.* As long as there are reinforced entries within the interest tables on relays along data path, the flow of data can continue.

If a sink does not receive data it requested after a certain span of time, it may recommence *interest* propagation, and the process will start again. The series of actions starting with *interest* propagation and the series of *data* and *reinforcement* transmissions up to next recommencement of *interest* propagation constitute an *interest cycle.*

3.1.5 Sink Logic

A sink may need to perform a certain application specific computation, which requires one or the other data type. It could also be the case that sink requires different data items for processing. In the first case a sink can use Or logic for data selection, where as in the second case it uses And logic.

In both cases, during the *interest* propagation phase the sink sends out interest for all data types. Depending on limitations of packet lengths in the wireless sensor network application, the sink could pack as many *interest*s as possible into a single message. This *interest* message is called an *aggregate interest.*

Whenever a relay receives an aggregate interest message, it needs to do processing for each of the encompassed data types. The processing may be done in a fashion as if *interest* for each data type were received separately. If an interest message has to be forwarded, then it can be forwarded again in an aggregate interest message, but only those data types whose interests need to be forwarded will be included in this aggregate interest message. Sources also process aggregate interest messages in the same ways as relays, except it need not forward it.

When a sink receives the first data message for a particular type it requested after *interest* transmission, its response depends on the logic used. If it uses Or logic, then it responds only to one of the data messages it receives with a reinforcement message even if it receives other *data* before the start of next refresh cycle. At the start of next refresh cycle, the neighbor that delivered the least cost data among all data types sink has received, will be transmitted a reinforcement.

When a sink using And logic receives a data type it is interested in, it waits for a period before it will send out *reinforcement* against the least cost data message that it received for that particular data type. At the start of the next *refresh cycle* for each data type sink received data message against its *interest*, the sink transmits *reinforcement* to the neighbor that delivered the least cost data message.

3.1.6 Aggregation

Data aggregation: It can be achieved at relay nodes by combining *data* of different data types whenever they are to be forwarded to same neighbor. In Figure 3.3 sink C1 requires both data types produced by source P1 and P2. At relay node 5, which is the starting point of common path shared by both data types, data aggregation could be done.

Interest aggregation: It is very much possible that several sinks may be in the *interest* propagation phase at the same time. Thus there is a chance that relays may receive distinct *interest*s with the scope of aggregation. When a relay receives different interest messages

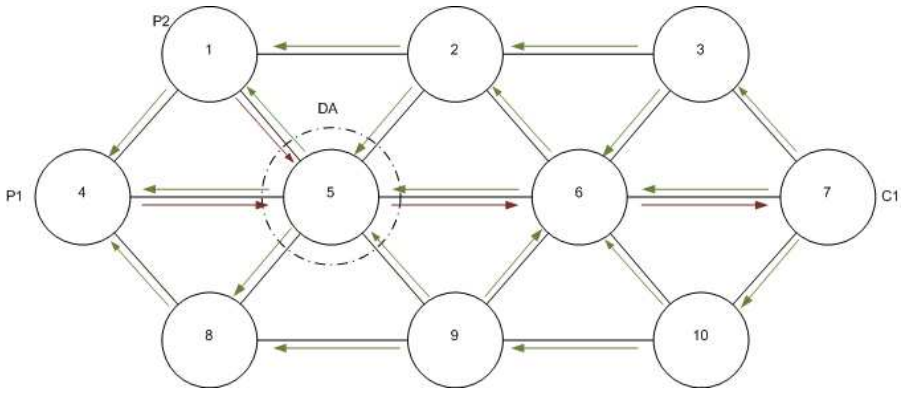


Figure 3.3: *An example for data aggregation*

from the same sink for the same data type, then only the one with the lower cost needs to be forwarded. Also in the case when more than one *interest* are at a relay that need to be forwarded they may be aggregated.

3.2 Implementation

The algorithm is implemented in nesC for TinyOS platform and is simulated on TOSSIM. The components are implemented in such a way that *data* and *interests* are handled in separate modules. Algorithms for source, sink and relays are discussed separately in the following discussion.

3.2.1 Messages

From the discussion so far, it is clear that there will be three different types of messages flowing in network, viz., *interest, data and reinforcement.* Interests messages are identified by the sink that generated the *interest* and the data type that sink is interested in. The message must also bear the identifier of the node that transmitted it as *interest* progress through the network, as well as the cost of path traveled thus far.

Reinforcements also have the same requirement as *interests.* Thus both these messages can have same format and there is a need for a field in these messages to identify whether they are *reinforcement* or *interest* or *data.* The messages also require a field to identify the

newness of the message. This can be implemented with a sequence number.

Format of the *interest* message is as follows:

- MSG_TYPE : Identifies the message as *interest*. Length is 1 byte with value 0x01
- $PREV\_HOP$: Identifies the node that transmitted this message. Length is 2 bytes
- MSG_SRC_ID : Identifies the sink requesting data. Length is 2 bytes.
- $DATA\_TYPE$: Identifies the type of data requested by sink. Length is 1 byte.
- $COUNT$: Gives the message number at the sink. Length is 2 bytes
- $COST$: Gives the cost of interest message. Length is 2 bytes.

The reinforcement message format is given below:

- MSG_TYPE : Identifies the message as *reinforcement*. Length is 1 byte with value 0x02
- $PREV\_HOP$: Identifies the node that transmitted this message. Length is 2 bytes
- MSG_SRC_ID : Identifies the sink requesting data. Length is 2 bytes.
- $DATA\_TYPE$: Identifies type of data requested by the sink. Length is 1 byte.
- $COUNT$: Gives the message number at the sink. Length is 2 bytes
- $COST$: Gives the cost of reinforcement message. Length is 2 bytes.

In addition to the fields described above, data messages will have a field to hold the actual data. The data message structure is as follows:

- MSG_TYPE : Identifies the message as *data*. Length is 1 byte with value 0x03
- $PREV\_HOP$: Identifies the node that transmitted this message. Length is 2 bytes

- MSG_SRC_ID : Identifies the sink requesting data. Length is 2 bytes.
- $DATA\_TYPE$: Identifies the type of data in the message. Length is 1 byte.
- $COUNT$: Gives the message number at the sink. Length is 2 bytes
- $COST$: Gives the cost of data message. Length is 2 bytes.
- $VALUE$: This field contains data. Length is 2 bytes

The cost used in this implementation is hop count multiplied by ten. This is to accommodate integer division for the cost division heuristic.

Interests are broadcast where as *reinforcements* are unicast, so they can not be aggregated. However, they can be aggregated separately. An aggregate message is identified by its first field. The next field will contain the number of *interests* that are packed in the aggregate message, followed by a field for previous hop. This is followed by units of *interests* consisting of a sink identifier, data type, cost of *interest* and count. In a similar manner *reinforcements* can also be packed. In addition to the fields discussed for other aggregate messages, data aggregate messages require the field with data as well to be packed in the aggregate data message.

3.2.2 Tables

Every node in the network maintains both a data and an interest table. Fields in the data table correspond to parameters in data messages. The data table is meant to cache data messages. A timestamp is associated with each entry in table. A timer is used to check for the expiration of entries. The structure of the table is given in 3.1.

$SOURCE$	$TYPE$	$COUNT$	$DATA$	$COST$	$PREV\_HOP$	$STAMP$

Table 3.1: *Data table format*

The interest table is used to cache *interest* and *reinforcement* entries. Thus the fields of the table correspond to interest and reinforcement messages. As it was discussed, an

interest message is identified by a sink and data type, correspondingly there will be only one entry per sink and data type. A timestamp is also associated with each entry.

SOURCE	*COUNT*	*TYPE*	*REINFORCED*	*COST*	*PREV_HOP*	*STAMP*

Table 3.2: *Interest table format*

The tables are maintained in separate modules. Care must be taken in adjusting the timer parameters such that when a reinforcement message arrives at a relay there is still data in the data table. The timestamp is simply a count which marks the epoch in which a table entry expires. Thus whenever that mark is reached that entry is removed. As discussed in the design phase, whenever an update is made in the entry the timestamp for expiration is updated to extend the entry's life.

3.2.3 Sink Node Algorithm

A sink node maintains two timers, one for *interest cycles* and another for *refresh cycles*. The timer maintaining *interest cycles* is called the *interest* timer and timer maintaining *refresh cycles* is called the *reinforcement* timer.

Code Listing 3.1: *Sink Algorithm - Broadcast interest*

```
Node * p1;
MoteDataType * q;
GenMessage msg;

inline void updateIntMessage() {
    sendmsg.msg_type = IntType;
    sendmsg.count = count;
    sendmsg.cost = 10; // for 1 hop, cost = 1*10
    sendmsg.msg_src_id = TOS_LOCAL_ADDRESS;
    sendmsg.prev_hop = TOS_LOCAL_ADDRESS;
}

task void sendInterest() {
    atomic q = moteInfo[TOS_LOCAL_ADDRESS]->dType;
    while( q != NULL ) {
        updateIntMessage();
        sendmsg.data_type = q->type;
```

```
        call GenMsgOutput.output(&sendmsg, TOS_BCAST_ADDR);
        q = q->next;
    }
}
```

When a sink starts up, it waits for some time and then it starts its interest timer. It then enters *interest* propagation phase where it broadcast *interests* of all data types that it has interest. After broadcasting *interests*, the sink waits for responses. Data tables are updated on the arrival of each data message. When the first data message in response to an *interest* arrives, the *reinforcement* timer is started and a reinforcement message is sent.

As described previously, depending on sink logic, further arrival of data messages may or may not result in the transmission of *reinforcements*. In the case of a sink that consumes only one data type, no further transmission of *reinforcements* occur until the beginning of the next *refresh cycle.*

Code Listing 3.2: *Sink Algorithm - Reinforcement transmission*

```
bool type[NUMTYPES]; /* initialized to FALSE at
                        start of interest propagation phase*/

bool trigger() {
    bool value = FALSE;
    int i = 0;
    while(i < moteInfo[TOS_LOCAL_ADDRESS]->ntypes) {
        value |=type[i];
        i++;
    }
    return value;
}

event DataMessage * InterestData.receivedData(DataMessage * m) {
    int i = 0;
    //start reinforcement timer
    if(!trigger() )
        call ReinforceTimer.start(TIMER_REPEAT, RefreshPeriod );
    q = moteInfo[TOS_LOCAL_ADDRESS]->dType;
    while(i < moteInfo[TOS_LOCAL_ADDRESS]->ntypes || q != NULL) {
        if(m->data_type == q->type && !type[i]) {
            if( moteInfo[TOS_LOCAL_ADDRESS]->logic == AndLogic )
                reinforce(q->type);
            else if( !trigger() )
```

```
                reinforce(q->type);
            type[i] = TRUE;
        }
        i++;
        q = q->next;
    }
    return m;
}
```

If a sink uses Or logic, only the first data message arrived will result in *reinforcement* until the next *refresh cycle*. In the case of And logic, at the arrival of first data message of each data type that sink is interested in, a reinforcement message will be unicast to the relay that delivered it. A Boolean array is used to save the status of the reception of data message for each data type. This is illustrated in listing 3.2. When its *reinforcement* timer expires, the sink sends a reinforcement message. If the sink consumes only one data type, which is the case when it uses Or logic or it has interest only on one data type, *reinforcement* is sent only to the neighbor that delivered the least cost data message. In the case of Or logic, the neighbor that delivered the least cost data among all the data types sink has interest, will be sent *reinforcement*.

Code Listing 3.3: *Sink Algorithm - Refresh Timer expiration*

```
GenMessage sendmsg;

inline result_t reinforce(uint16_t t) {
    uint16_t hop = call DataTableI.getLeastCostHop(t);
        atomic {
        sendmsg.prev_hop = TOS_LOCAL_ADDRESS;
        sendmsg.msg_src_id = TOS_LOCAL_ADDRESS;
    }
    sendmsg.count = ++count;
    sendmsg.msg_type = ReInfType;
    sendmsg.data_type = t ;
    sendmsg.cost = call RouteTableI.getCost(hop);
    return call GenMsgOutput.output(&sendmsg, hop );
}

event result_t ReinforceTimer.fired() {
    int i=0;
    if(moteInfo[TOS_LOCAL_ADDRESS]->logic == AndLogic) {
```

```
        q = moteInfo[TOS_LOCAL_ADDRESS]->dType;
        while(q != NULL) {
            if(type[i++])
                reinforce(q->type);
            q=q->next;
        }
    } else if(moteInfo[TOS_LOCAL_ADDRESS]->logic == OrLogic) {
        return reinforce( call DataTableI.getLeastCostDataType(
            moteInfo[TOS_LOCAL_ADDRESS]->dType) );
    }
    return SUCCESS;
}
```

In the case of And logic, *reinforce* is sent for each data type to the neighbor that delivered the least cost data of that type. In the case of a simple sink that consumes only one data item, it is set to use And logic, with the number of data types one.

A *Reinforcement* timer is set up as a repeat timer, so that once it expires it will restart itself. When the *interest* timer expires, the *reinforcement* timer is stopped. The sink enters *interest* propagation phase, i.e., the beginning of the algorithm, and the process repeats.

3.2.4 Relay node algorithm

Every relay maintains an interest table to cache interest messages and a data table to cache data messages. When a relay receives an interest, it consults the interest table. If there is no interest in the table with same data type for that sink, a new entry is made. If there is an entry, then that entry is updated only if it is a newer interest message or it is a duplicate interest message with lower cost. The rule to propagate will appropriately set a flag to indicate that an interest message may or may not be propagated. The rule might be such that a duplicate interest message need not be forwarded. The other rule that is implemented is that duplicate interest messages with lower cost will be forwarded.

In Figure 3.4 nodes P1 and P2 generate the same type of data and nodes C1 and C2 are consumers of this data. Path 1 is a shortest path between producer P2 and consumer C2, where as path 3 is the shortest path between producer P1 and consumer C1. The overall cost of transmitting a data message through these two paths are higher than the shared

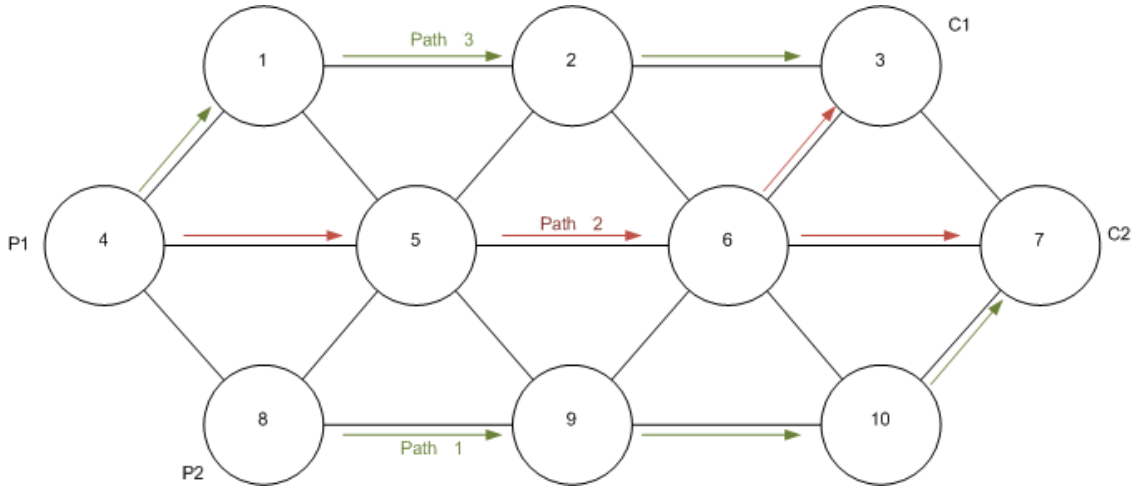


Figure 3.4: *Cost division*

path 2(here cost is simply the number of hops). C1 and C2 can be made to select a path through node 6, if it has lower cost than paths 3 and 1 respectively. This can be made possible by cost division approach. If the cost(which is 2) of data path from node 4 to 6 is distributed into branches, then the cost($2/2 + 1$) of path through node 6 will be lower than the cost(which is 3 each) of path 1 and 3.

If at reception of an interest message, it is identified as new and data is already available, then *data* is forwarded to neighbor that delivered the interest message. For example, consider that the path between nodes P1 and C1 via nodes 5 and 6 is already reinforced when node C2 sends out *interest*. Node 6 need not further propagate *interest* from C2 as data is already available and hence only replies with data with the cost distributed over existing reinforced path and the new path to node C2(this is the implementation of greedy incremental heuristic). The new cost for the data message that will be forwarded can thus be computed from the expression(but only if cost division heuristic is implemented):

$$new\ cost = (old\ cost/(\#reinforced\ paths\ for\ data\ type + 1)) + cost\ of\ next\ link$$

Code Listing 3.4: *Relay Node Algorithm - Interest processing*

```
event GenMessage * ReceiveGenMsg.receive(GenMessage * message){
```

```
    if(message->msg_type == IntType ) {
        if( !consume(message->data_type) ) {

            intTable[TOS_LOCAL_ADDRESS] = call IntTableI.updateEntry(
                    intTable[TOS_LOCAL_ADDRESS], message);
            if(!produce(message->data_type) ) {
                if(intProcessing[TOS_LOCAL_ADDRESS] == TRUE) {
                    atomic intProcessing[TOS_LOCAL_ADDRESS] = FALSE;
                    if( (call DataTableI.getLeastCost(
                                message->data_type)) > 0 ) {
                        signal DataInterest.sendData(message->prev_hop,
                                message->data_type);
                        atomic propInt[TOS_LOCAL_ADDRESS] = FALSE;
                    }
                    if( propInt[TOS_LOCAL_ADDRESS] == TRUE &&
                            !consume(message->data_type) ) {
                        propogateInterest(message);
                        atomic propInt[TOS_LOCAL_ADDRESS] = FALSE;
                    }
                }
             }

                ...
    return message;
}

event result_t DataInterest.sendData(uint16_t dest, uint8_t d) {
    call DataTableI.getDataMsg( &senddata, d );
    atomic senddata.prev_hop = TOS_LOCAL_ADDRESS;
    senddata.cost = senddata.cost/(computesize(call
            IntTableI.getReinfNeighbors(dType)) + 1);
    senddata.cost += call RouteTableI.getCost(dest);
    return call DataOutput.output(&senddata, dest);
}
```

[1]

When a relay receives a data message, the data table is consulted. If it is already in the table, the packet is dropped. If there is an entry for data, but *data* received is new then the data message is forwarded and the entry is updated by modifying its time stamp to extend its expiration time. If the data message is flagged to be forwarded, then the following rules

[1]RouteTableI is an interface that maintains a list of neighbors of the node

are used to forward data: If it is the first data message of its type, the interest table is consulted to retrieve interested neighbors. These are the neighbors that delivered least cost *interests* for the data type. Since there is only one entry per sink and data type combination for each sink, there will be only one neighbor per interest entry in this list of neighbors. A data message is then unicast to each of these neighbors. The cost of the data message is distributed equally over the new paths if the cost division heuristic is implemented. In Figure 3.4, when a data message arrives at node 6 for the first time and sees the distinct *interests* from nodes 3 and 7, it will forward the data message to both neighbors and the cost of the *data* thus forwarded may be computed(again if the cost division heuristic is implemented) with the expression given:

$$new\ cost = (old\ cost/\#interested\ sink\ node\ paths) + cost\ of\ next\ link$$

Code Listing 3.5: *Relay Node Algorithm - Data Message Processing*

```
event DataMessage * ReceiveDataMsg.receive(DataMessage * m) {
    if( call DataTableI.getLeastCost( m->data_type) == 0 && !consume(
            m->data_type) ) {
        //no entry in Data table for the data type
        list2 = call IntTableI.getIntNeighbors( m->data_type ) ;
    } else {
        list2 = call IntTableI.getReinfNeighbors( m->data_type ) ;
    }
    listsize = computesize(list2);
    dataTable[TOS_LOCAL_ADDRESS] = call DataTableI.updateEntry(
            dataTable[TOS_LOCAL_ADDRESS], m);
    while(list2 != NULL &&  propData[TOS_LOCAL_ADDRESS]) {
        if(list2->id != m->prev_hop ) {
            memcpy(&rcvdata, m , sizeof(DataMessage));
            atomic rcvdata.prev_hop = TOS_LOCAL_ADDRESS;
            rcvdata.cost = m->cost/listsize + call RouteTableI.getCost(
                        list2->id);
            call DataOutput.output(&rcvdata, list2->id );
        }
        list2 = list2->next ;
    }
    return m;
}
```

If the data message is to be forwarded and it is not the first data message of its type to reach the relay, then it is forwarded only to the neighbors that delivered reinforcement messages. Again the data message is unicast to each of these nodes. For example, when data paths to nodes 3 and 7 are confirmed through node 6, and *data* reaches node 6, each of the forwarded messages from node 6 can have the cost of data message distributed over the paths(if cost division heuristic is implemented) with the expression given below.

$$new\ cost = (old\ cost/\#reinforced\ paths\ for\ data\ type) + cost\ of\ next\ link$$

When a relay receives reinforcement messages, its interest table is updated to reflect that the *interest* has been reinforced(all the parameters of the entry are set to that of the one in the message). The reinforcement message is propagated as unicast to the neighbor that delivered the least cost data of the data type contained in reinforcement message.

Code Listing 3.6: *Relay Node Algorithm - Reinforcement Processing*

```
event GenMessage * ReceiveGenMsg.receive(GenMessage * message){
    if(message->msg_type == IntType ) {
            ...
    } else {
        intTable[TOS_LOCAL_ADDRESS] =  call IntTableI.updateEntry(
                intTable[TOS_LOCAL_ADDRESS], message );
        if(!produce(message->data_type) && propInt[TOS_LOCAL_ADDRESS]){
            atomic next_hop = call DataTableI.getLeastCostHop(
                        message->data_type);
            memcpy(&sendmsg, message, sizeof(GenMessage));
            atomic sendmsg.prev_hop = TOS_LOCAL_ADDRESS;
            sendmsg.cost = message->cost + call RouteTableI.getCost(
                        next_hop);
            call GenMsgOutput.output(&sendmsg, next_hop );
        }
    }
    ...
    return message;
}
```

3.2.5 Source node algorithm

Every source maintains an interest table similar to relay node. When a source node receives an interest message for data it produces, it consults the interest table. If there is no entry, then it makes a new entry. If there is an entry and the interest message is a duplicate, it updates to reflect the lower cost. Otherwise it is ignored. It sends *data* to its neighbor whenever an update is made in the interest table. When it receives reinforcement messages, the source just updates the interest table.

Code Listing 3.7: *Source Node Algorithm - Interest/Reinforcement Processing*

```
event GenMessage * ReceiveGenMsg.receive(GenMessage * message){
    if(message->msg_type == IntType ) {
        if( !consume(message->data_type) ) {
            intTable[TOS_LOCAL_ADDRESS] = call IntTableI.updateEntry(
                    intTable[TOS_LOCAL_ADDRESS], message);
            ...
        } else {

            if( ! call IntTableI.hasCheaperInt(message->msg_src_id,
                    message->data_type, message->cost) )
                signal DataInterest.sendProduce(message->prev_hop,
                                message->data_type );
        }
    } else {
        intTable[TOS_LOCAL_ADDRESS] = call IntTableI.updateEntry(
                intTable[TOS_LOCAL_ADDRESS], message );
        ...
    }
    return message;
}
```

Source nodes use a timer to periodically send data messages for reinforced data requests. This timer is started when the node receives its first interest message. For every reinforcement message received, the interest table is updated to save *reinforcement* from the data requester. When the data timer expires, the source checks the interest table for all reinforced interest requests and sends *data* for them. For each consumer requesting a data type, a data message is transmitted to a neighbor node that has the shortest route to the consumer.

Code Listing 3.8: *Source Node Algorithm - Timer expiry*

```
event result_t DataInterest.sendProduce(uint16_t dest, uint8_t dType) {
    // for interest in data produced by the node
    if(!firstData) {
        firstData = TRUE;
        call DataTimer.start(TIMER_REPEAT, DataPeriod );
    }

    producedata.data_type = dType;
    producedata.msg_src_id = TOS_LOCAL_ADDRESS;
    producedata.prev_hop = TOS_LOCAL_ADDRESS;
    producedata.msg_type = DataType;
    producedata.count = ++count;
    producedata.value = data;
    producedata.cost = call RouteTableI.getCost(dest);
    return call DataOutput.output(&producedata, dest);
}

event result_t DataTimer.fired() {
    return call ADC.getData();
}

async event result_t ADC.dataReady(uint16_t d) {
    atomic data = d;
    post send();
    return SUCCESS;
}

task void send() {
    atomic mdt = moteInfo[TOS_LOCAL_ADDRESS]->dType;
    count++;
    while(mdt != NULL ) {
        dt = mdt->type;
        atomic list = call IntTableI.getReinfNeighbors(dt);
        while(list != NULL ) {
            producedata.prev_hop = TOS_LOCAL_ADDRESS;
            producedata.msg_src_id = TOS_LOCAL_ADDRESS;
            producedata.msg_type = DataType;
            producedata.data_type = dt;
            producedata.value = data;
            producedata.count = count;
            producedata.cost = call RouteTableI.getCost(list->id);
            call DataOutput.output(&producedata, list->id );
            list= list->next;
        }
```

```
            mdt=mdt->next;
        }
}
```

3.2.6 Message Transmission

Every node in the network maintains separate queues for transmission of messages. Transmission of *interest* and *reinforcements* are handled by a separate module than the one which handles data messages.

In the module which handles *interests* and *reinforcement*, *interest* is given preference over *reinforcement*. There are separate queues maintained for each neighbor in which reinforcement messages are kept and one for broadcast in which interest messages are kept. When there is more than one message in the queue, messages can be aggregated. The limit of the aggregated packet depends on the mote type and its maximum packet length supported by the MAC layer. Currently the implementation sets limit such that a node can aggregate up to 4 interest/reinforcement messages, and up to 3 data messages. If there are older messages within the queue when a new message is queued up for delivery, the older one will be replaced by the latest one. Also in the module that handles data messages, there is no broadcast queue unlike the module which handles *interest*. This is because data messages are always unicast.

3.2.7 Heuristics

This section explains the various algorithms that have been implemented and are analyzed in the next chapter. All the algorithms implement data message aggregation.

Base Case: An algorithm which does not support any heuristics is implemented. It does not support interest aggregation, which means if a sink has interest in four data types, it will send out four different interest messages.

Whenever a relay receives an interest message for a data type, it is forwarded only if it is a new interest. As described previously a new interest message is identified by comparing its

parameters with entries in the interest table. The algorithm uses a 16 bit sequence number to determine the freshness of a message. Sequence wrap happens at a relay for an entry in the interest table whenever it sees a message with a very low sequence number after seeing some of the highest sequence numbers. Same is the case for data messages. Algorithm also assumes that when a failed node restarts, it will have no previous state information.

Besides the base case does not implement shortest path heuristic either, which implies it does not forward lower cost duplicate packets either.

Heuristic 1: This is a modified version of the base case, which supports interest aggregation. As described in section 3.2.6, up to 4 interest messages can be aggregated.

Consider a simple application with one source and one sink, both producing and consuming two data items respectively. If interest messages are sent separately, there is a greater chance that different paths may be set up for each data type even if they are of same length. This will be illustrated later in Section 4.2.1. But if interest messages are aggregated, the probability of the path convergence is higher. This idea is extended to more than one producer and one consumer and their performance is analyzed in Chapter 4.

The interest aggregation takes place at relays where interest comes from diverse sinks as explained in Section 3.2.6. As the number of sinks and data types increase in the network, the constraint on the aggregation capability is limited by MAC layer. As obvious, this heuristic reduces the number of *interest* transmitted when compared to algorithm with no heuristics.

Heuristic 2: This is an improvement on heuristic 1 with forwarding of lower cost interest(the shortest path heuristic). It also supports the transmission of *data* in response to a new interest at a relay(greedy incremental heuristic), if *data* requested is available.

Although forwarding of lower cost interests as discussed in 3.2.4, adds an overhead, it gives a better metric to determine the data path. This will lead to selection of shorter data paths. This heuristic in addition to that of interest aggregation leads to a data graph with lower number of transmissions than the one obtained due to heuristic 1.

When a node in already established data path receives an interest message, it can reply with the data that is in store. Besides, the node does not forward interest messages, as the data path is already established.

Heuristic 3: This is an extension of heuristic 2 with support for cost division. Cost is divided at three different cases

1. Whenever an *interest* reaches an established path(reinforced path) to a different sink but for same data, the relay replies with *data*, whose cost is distributed between the new path and existing path/paths.

   The case when one sink completes *interest* propagation phase and enter *data* propagation phase followed by another sink and so on in a sequential manner will lead to data graph/tree established in the manner of a greedy incremental heuristic. Cost distribution as explained in Section 1.3 can lead to selection of a shared longer path over unshared shorter paths.

2. Whenever a data message reaches a relay node that does not have any entry in the data table for that data type, the cost of *data* is divided onto interest paths *data* is to be forwarded.

   As described before, the cost division approach is to favor shared data message paths over shorter unshared paths. Since a sink node sends *reinforcement* to the first data message arrived, the chance of selecting these shared longer paths are less on the first *refresh cycle*. But this path may be selected at next *refresh cycle*, unless a newer data message from the same source is received at the sink through a different path.

3. Whenever a data message reaches a relay and needs to be forwarded to more than one reinforced paths.

   This distribution of cost to reinforced paths, helps in maintaining the preference of the shared path during the first two cycles. After this period, the other data message

paths which were established as a result of *interest* propagation phase would have expired.

In the next chapter, the performance of these algorithms will be compared for various test cases set up.

Chapter 4

Testing and Analysis

4.1 Introduction

4.1.1 Test case assumptions

The algorithms were tested with assumption that no information regarding topology is available to any node in the network. Results are collected after running algorithm for each epoch of interest propagation phase followed by data flows from various sources to sink.

4.1.2 Network topology

Sources and sinks are placed in the test topologies where they are diagonally opposite in a square grid. The maximum number of links in a data tree made by an algorithm using shortest path heuristic, is the sum of all the shortest path lengths between sources and sinks. The minimum number of links in the data tree is the number of links in minimum Steiner tree formed by sources and sinks. From the test results it can be seen that these two values form a bound on the number of links on the graph formed by the algorithm. As the quality of trees are affected only by the traversing of messages in between source and sink nodes, nodes outside this space are not simulated.

The number of messages transmitted during exploratory phase may be assumed proportional to number of nodes in the test grid, where as the number of data messages is proportional to data tree, as data aggregation is used in all algorithms.

1P1C	1P2C	1P4C
2P1C	2P2C	2P4C
4P1C	4P2C	4P4C

Table 4.1: *Test cases for And logic*

Testing is done with nodes starting with square grid 6x6, whose shortest path between diagonally opposite nodes is of length 5. And the number of nodes simulated is 16, allowing 5 nodes above and below the diagonal. Testing is further done on nodes on square grids with dimensions increased in steps of 2. The next test grid in the sequence has 22 nodes from a 8x8 grid.

4.2 Test cases for Sinks with And logic

This set also includes the simple consumers. Testing were done on cases shown in table 4.1 where in each test case a producer will generate a distinct data type and each consumer requests for all data types produced in the network. For example in the case of 4P4C each of the four sources generate four different data items and each of the four consumers requests all of the four data types. And the sink uses And logic, so they require all data types at all times.

4.2.1 Effect of interest aggregation

In the algorithms modeled, when a node is delivered a message, the node considers its neighbor as the source of message. The only metric that a sink can use is the cost field to select a data path over another. The question how does interest aggregation help to select a better path which can have more scope for data aggregation is answered here.

Consider a network where a sink requires two types of data, say temperature and pressure readings and a source in the network generates both these data(See Fig 4.1 which is not a test case). When there is no interest aggregation, sink sends out request separately. This can cause the set up of different paths for data even if one source can supply both

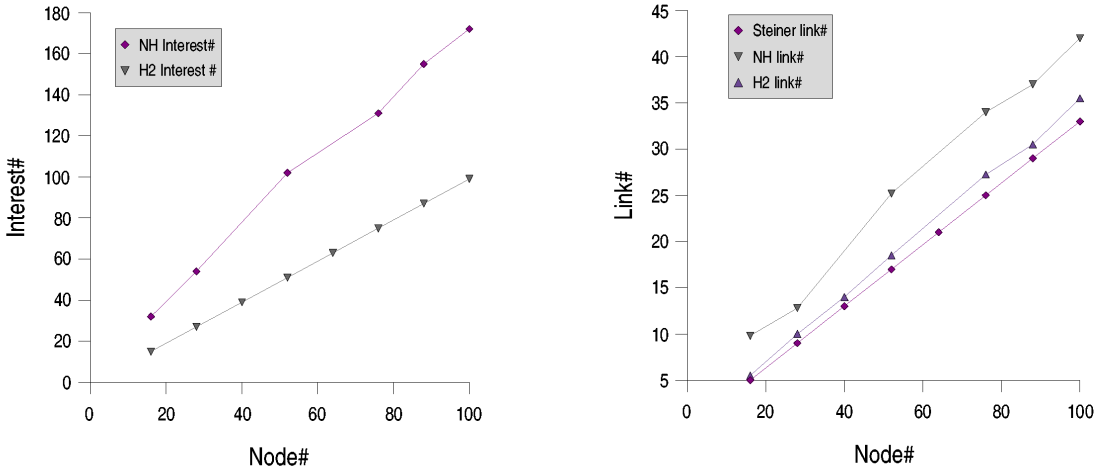


Figure 4.1: *Results - 1P1C2DAnd*

the data. Fig 4.1 shows the difference in number of interest messages transmitted without interest aggregation(NH) and with interest aggregation and forwarding of lower cost interest messages(H2). The improvement in the number of links in the data tree formed in H2 is illustrated in Fig 4.1.

Interest aggregation alone is added to the H1 heuristic in addition to the base case. Observations made on H1 test runs in relation to NH test runs are as follows. The test case of 1P1C(see Fig 4.2) for a single data type, number of interest messages transmitted in algorithms NH and H1 are the same as there is no scope for interest aggregation.

Hence, there is 0% improvement due to interest aggregation alone in the case 1P1C1D. For the case 1P2C1D(see Fig4.4) there is 7.07% decrease in the number of interest messages transmitted in algorithm H1 than in the algorithm NH. And in 1P4C1D(see Fig4.8), this difference is improved to 41%.

The test cases with two data items show a similar trend of improvement in the number of interest message transmitted as in the case of test cases with one data item. For the case 2P1C2D, the improvement of H1 over NH is 48.33% (see Fig 4.10). The test results of 2P2C2D show 53.72% improvement(see Fig 4.12) and that of 2P4C2D show 56.32%

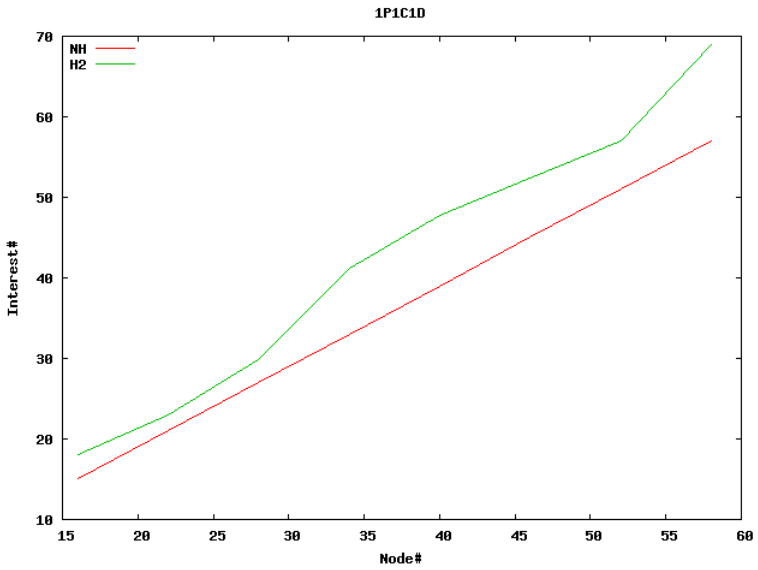


Figure 4.2: *Results - 1P1C1D Nodes vs Interests*

improvement(see Fig 4.14).

Results of the tests with four data items also show a reduced number of interest message transmission for algorithm H1. For the case 4P1C4D, 4P2C4D and 4P4C4D the improvement of algorithms H1 over NH is observed as 73.29%(see Fig 4.16), 69.44%(see Fig 4.18) and 68%(see Fig 4.20) respectively. In this particular case, the improvement due to aggregation has reached saturation, as H1 algorithm is implemented to aggregate up to four different interest messages. Hence for the cases with two and four customers requesting all the four different data types, the aggregation of all the interests are not possible at the intermediate nodes.

Another result is observed when comparing the number of links in the networks formed for transmission follows. For test case 1P1C1D, no interest aggregation is possible. So the difference in the number of links formed due to H1 is 0% better than that due to algorithm NH. For tests 1P2C1D and 1P4C1D, the difference between H1 and NH are observed as 12%(see Fig 4.5) and 7.97%(see Fig 4.9) respectively. Lowering of the values in the last two test cases, reflects the increased complexity in aggregation.

In the test results with two data types, H1 showed an improvement of 20.48%, 11.4%

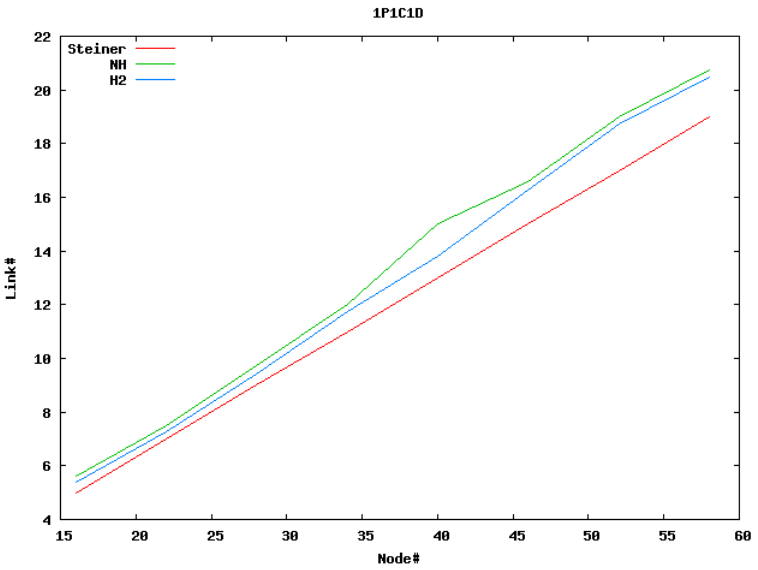


Figure 4.3: *Results - 1P1C1D Nodes vs Links*

and 5.35% for cases 2P1C2D(see Fig 4.11), 2P2C2D(see Fig 4.13) and 2P4C2D(see Fig 4.15)respectively over the algorithm NH. This result is consistent with the observations seen for cases 1P2C1D and 1P4C1D. And the results of cases 4P1C4D(see Fig 4.17), 4P2C4D(see Fig 4.19) and 4P4C4D(see Fig 4.21) with four different data types showed an improvement of 13.14%,10.1% and 6.57% respectively over NH. These results show that improvement due to H1 is inversely affected by the complexity of the network configuration.

4.2.2 Effect of forwarding lower cost interests

Test results for case 1P1C1D(see Fig 4.2) shows that the number of interests generated by algorithm with heuristic 2(H2) is higher than that of one with no heuristics(NH) by 17%. This is due to the lack of possibility of interest aggregation. For cases 1P2C1D(see Fig 4.4) number of interest messages in 12.58% higher than that of NH. Test results of case 1P4C1D(see Fig 4.8), the number of interest messages generated in H2 is 35% lower than that in NH. The two factors coming to play here are the reduction of messages due to integration, and increase in transmission of messages, due to propagation of lower cost interests introduced in H2. It can be see that, the reduction in number of messages due to

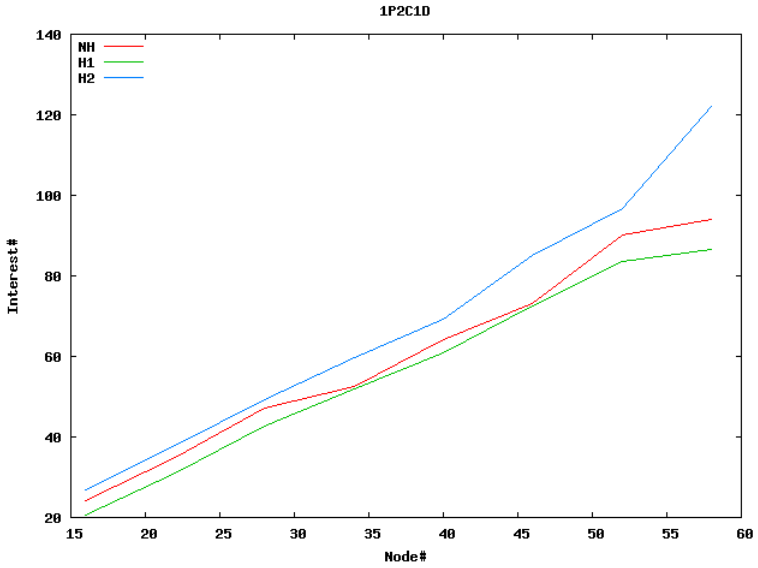


Figure 4.4: *Results - 1P2C1D Nodes vs Interests*

integration plays a higher hand in the case 1P4C1D.

Test cases 2P1C2D(see Fig 4.10), 2P2C2D(see Fig 4.12) and 2P4C2D(see Fig 4.14) show an improvement of 30.12%, 41.97% and 50.48% reduction in number of interest messages in H2 over NH. This clearly illustrates that, the effect of the increased number of messages due to shortest path heuristic is dwarfed by the reduction of the number of interest messages by interest aggregation.

The results of H2 test cases with four data types, 4P1C4D(see Fig 4.16), 4P2C4D(see Fig 4.18) and 4P4C4D(see Fig 4.20) have improvements of 66.91%, 57.69% and 66.14% respectively over NH. The aggregation has reached saturation levels for these test cases with four data types, as the limit is set to four for aggregation in this implementation, and interests from different consumers for same data type are treated as different interests.

When comparing the number of links in the network formed due to H2 and that due to NH, improvements have been observed due to the heuristics. Test cases 1P1C1D(see Fig 4.3), 1P2C1D(see Fig 4.5) and 1P4C1D(see Fig 4.9) shows improvement of 3.15%, 18.76%, and 11.83% respectively. It can be noticed that the case 1P1C is an instance of shortest path algorithm, which explains the low value in difference. For 1P2C and 1P4C, the percentage

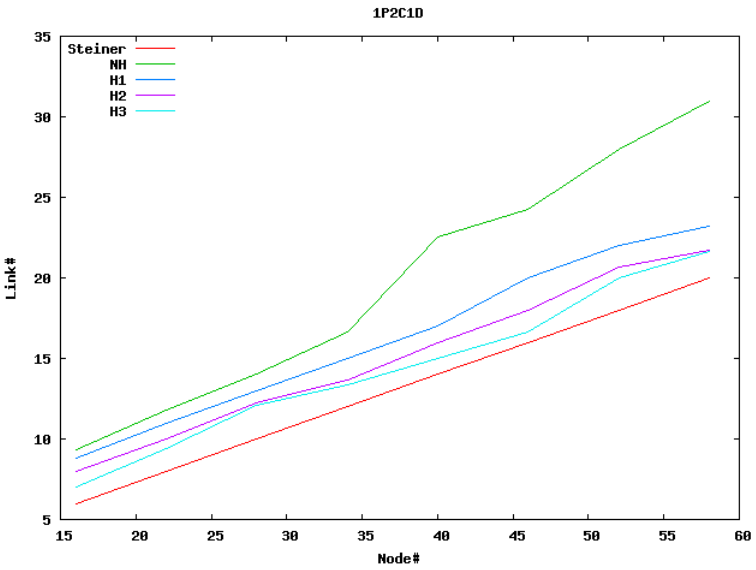


Figure 4.5: *Results - 1P2C1D Nodes vs Links*

reduction in link numbers is an indication of increasing difficulty of finding the minimal cost network.

The test results showed a reduction of 28.85%, 18.47% and 9.44% in the number of links in the transmission network formed due to H2 over NH for the cases with two data types 2P1C2D(see Fig 4.11), 2P2C2D(see Fig 4.13) and 2P4C2D(see Fig 4.15) respectively. The trends in the decreasing values in improvement shows the inverse affect of the complexity of network configuration.

A reduction of 25.7%,13.79% and 15.73% is observed for test cases 4P1C4D(see Fig 4.17), 4P2C4D(see Fig 4.19) and 4P4C4D(see Fig 4.21) in H2 when compared to NH. In this particular set of results, the saturation of interest aggregation affects the performance. The cases with four data types are border conditions, as interest aggregation is set a limit of four messages, where as for the data aggregation it is set to three. However, on increasing these aggregation limits to eight for interest aggregation and four for data aggregation, the reduction were observed for cases with four data types as 33%, 21.78% and 25.69% for cases 4P1C, 4P2C and 4P4C respectively. The improvement in the reductions can be attributed to better aggregation.

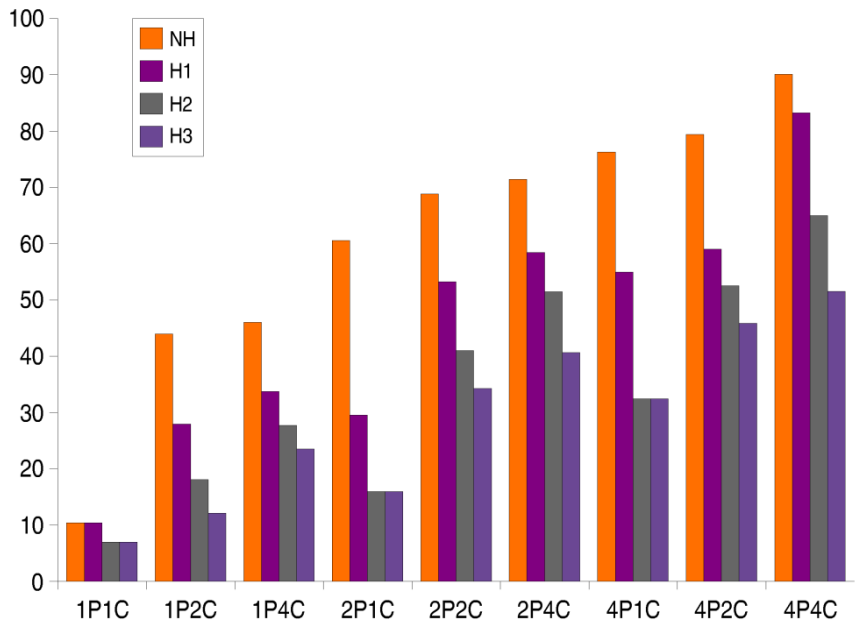


Figure 4.6: *Difference - Steiner tree and the result trees in percentages*

4.2.3 Effect of cost division heuristics

Algorithm with heuristics 3(H3) is not run for test cases with one sink(1P1C, 2P1C and 4P1C) as there is no room for cost division in these cases. Also there is no change in interest propagation phases of H3 and that of H2, so number of interests transmitted in H3 is not shown in the results.

The test case with a single data type 1P2C1D shows that the network formed during H3 is 22.82% better than NH. An improvement of 14.81% over NH is observed for test case 1P4C1D.

An improvement of 22.31% and 15.97% is observed in H3 run for test cases with two data types 2P2C2D and 2P4C2D over NH. Test runs of H3 in cases 4P2C4D and 4P4C4D shows a reduction of 17.47% and 22.58% respectively over NH. As observed previously, the test cases with four data type form the border test case, since the aggregation limit was set to four messages for interest aggregation. However, when this limit was increased to eight messages the reduction for cases 4P2C is improved to 27.26% and that for 4P4C it is

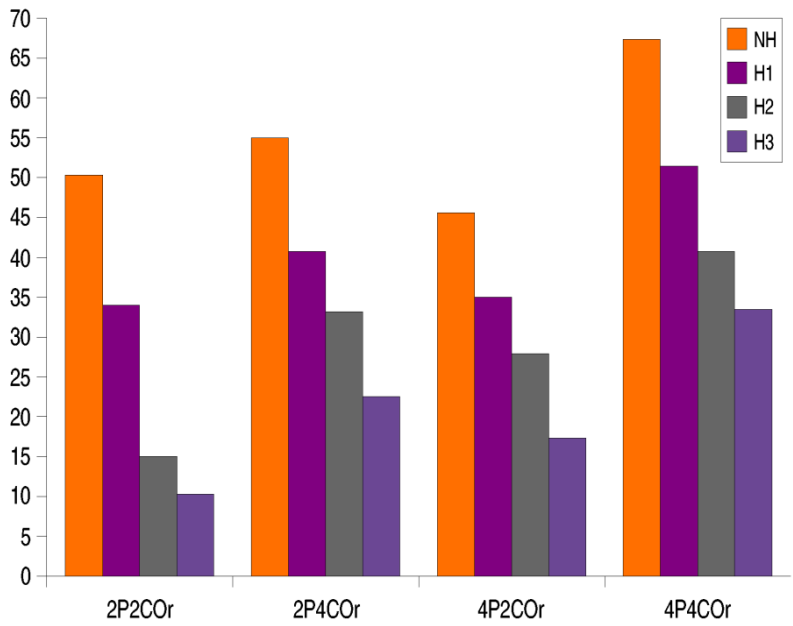


Figure 4.7: *Difference - Steiner tree and the result trees in percentages*

28.46%. It can also be noticed that values are closer.

Another observation that can be seen about the quality of data transmission paths from the graphs is that as the complexity of aggregation increases, the number of links in the graphs also tends to increase and move further away from Steiner tree. This is illustrated in Fig 4.6. It can also be observed that for each test cases, the communication graph generated by H3 performs better than H2, and H2 better than H1 and H1 better than NH. In other words, the trend that can be observed is, for the test case with one data types, as the number of consumer increases, the difference between Steiner tree and the result graph in terms of percentage change in the number of links increase. This observation is repeated for test cases with two data types as well as four data types.

4.3 Test cases for Or logic

Test cases for Or logic are shown in table 4.2. In each of these cases each source produce a distinct data item and each sink are interested in all the data types generated within the

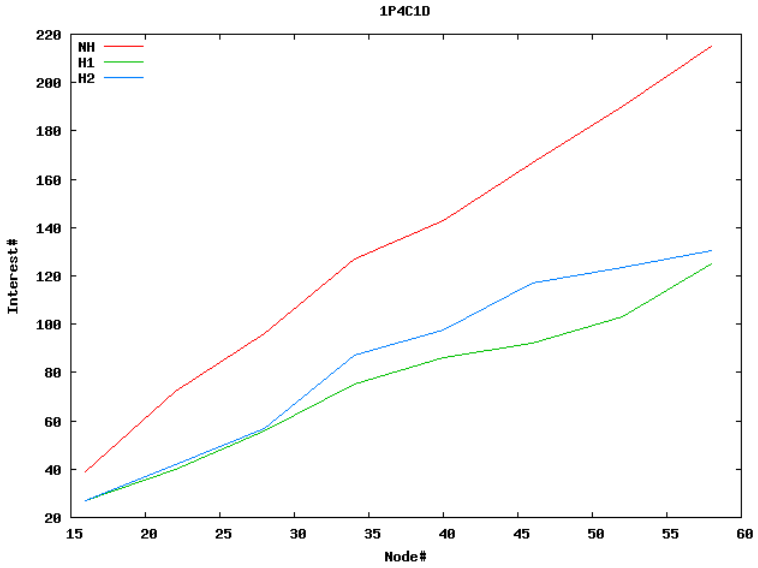


Figure 4.8: *Results - 1P4C1D Nodes vs Interests*

2P2C	2P4C
4P2C	4P4C

Table 4.2: *Test cases for Or logic*

network. Test cases with one source(1P) need not be simulated as there is only one data item, their performance will be the same as shown before. Test cases with one sink will be reduced to simply the case 1P1C, so they are not repeated either. The interest propagation phase is the same irrespective of the sink logic used. So they are not repeated in the results shown.

The graph shown in Fig 4.7 illustrates the relative quality of the communication graph generated in each test case with respect to the Steiner graph for that configuration. For each of the test case(see Figs 4.22, 4.23, 4.24, 4.25), the difference in quality of graph is computed by averaging the differences of each test result. Test case 2P2C has 50.33%, 34%, 14.98% and 10.25% difference with Steiner graph in terms of number of links for the runs of NH, H1, H2 and H3 algorithms respectively. For 2P4C these values become 54.97%, 40.74%, 33.19% and 22.53%. It can be seen, from these two tests that as the complexity of the configuration

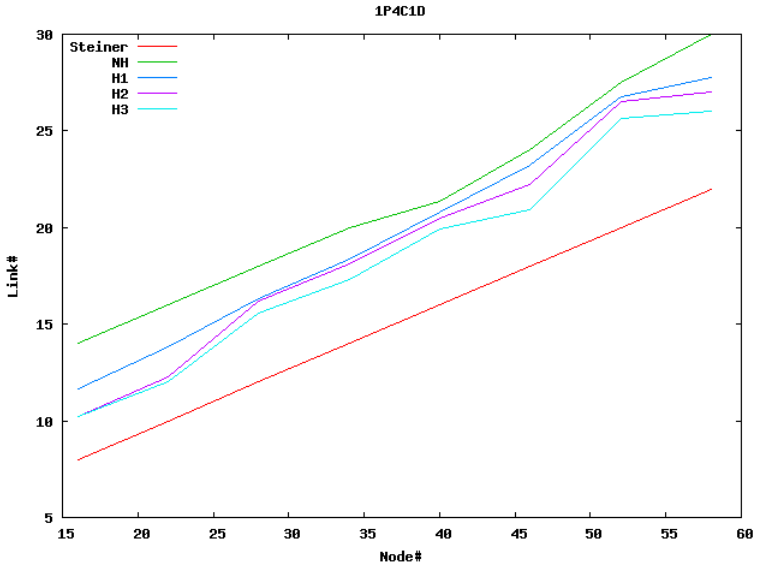


Figure 4.9: *Results - 1P4C1D Nodes vs Links*

increases, the deviation from optimal results also increases and H3 heuristics perform the best.

Test case 4P2C shows difference of 45.58%, 35%, 27.91% and 17.32% with Steiner graph in terms of number of links for runs of NH, H1, H2 and H3 respectively. 67.36%, 51.41%, 40.72% and 33.46% are the respective values for test case 4P4C. Here too, the observation held for the cases 2P2C and 2P4C holds. In short, in Or test results as well heuristic 3 performed better than heuristic 2, which in turn performed better than H1, as observed in the test results of And cases.

4.4 Space complexity

If s is the number of sinks in the network, and d is the total number of data types, then space required for interest table will be in $O(sd)$. But as the algorithms reach steady state, number of entries will be in $O(d)$ for each node on the data path.

If p is the number of sources in the network, then the space required for data table will be in $O(pd)$ at each node. When algorithms reach steady state, number of entries in the

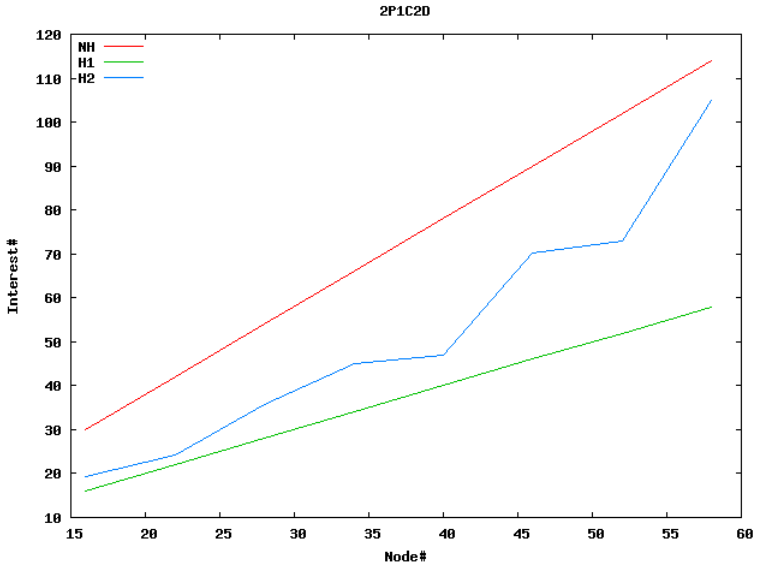


Figure 4.10: *Results - 2P1C2DAnd Nodes vs Interests*

table will be in $O(d)$ for each node on the data path.

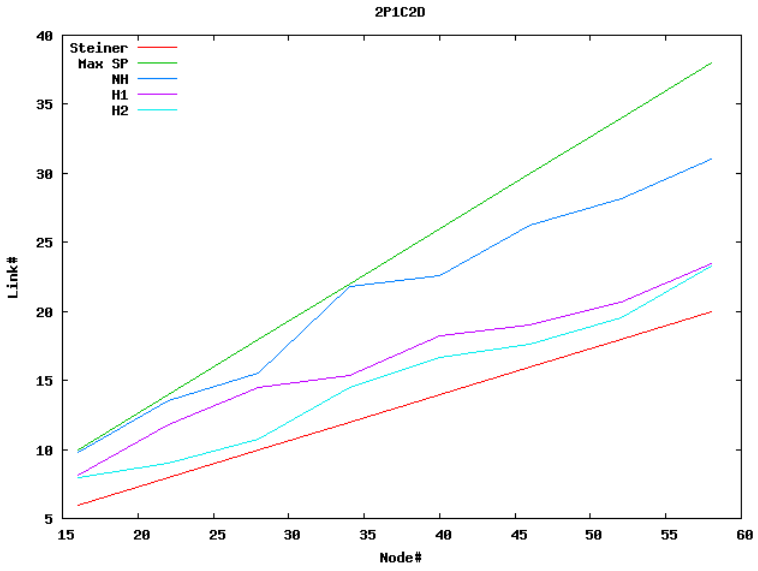


Figure 4.11: *Results - 2P1C2DAnd Nodes vs Links*

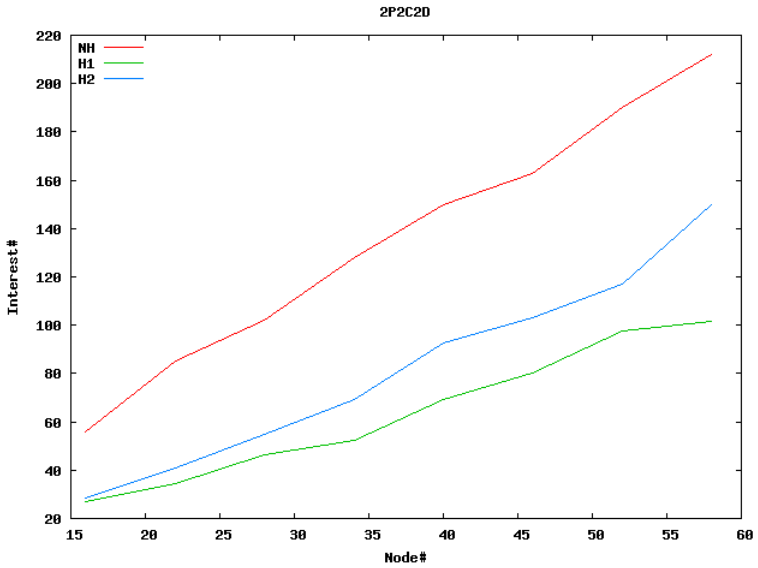


Figure 4.12: *Results - 2P2C2DAnd Nodes vs Interests*

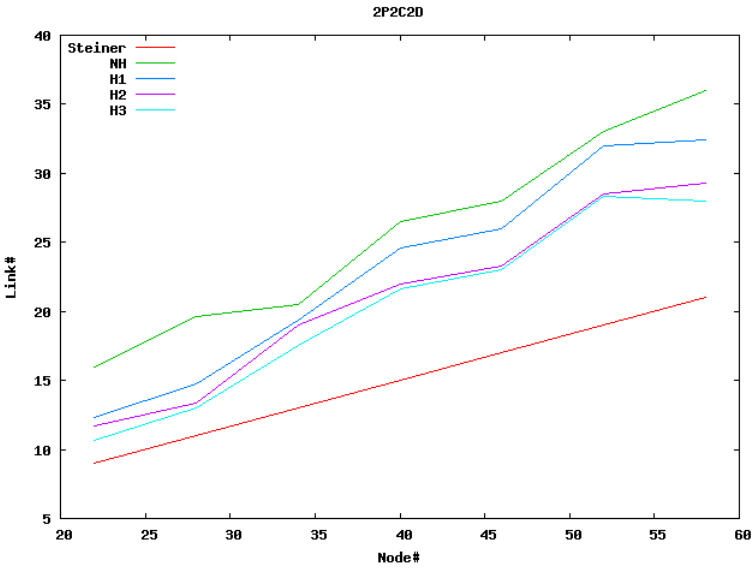


Figure 4.13: *Results - 2P2C2DAnd Nodes vs Links*

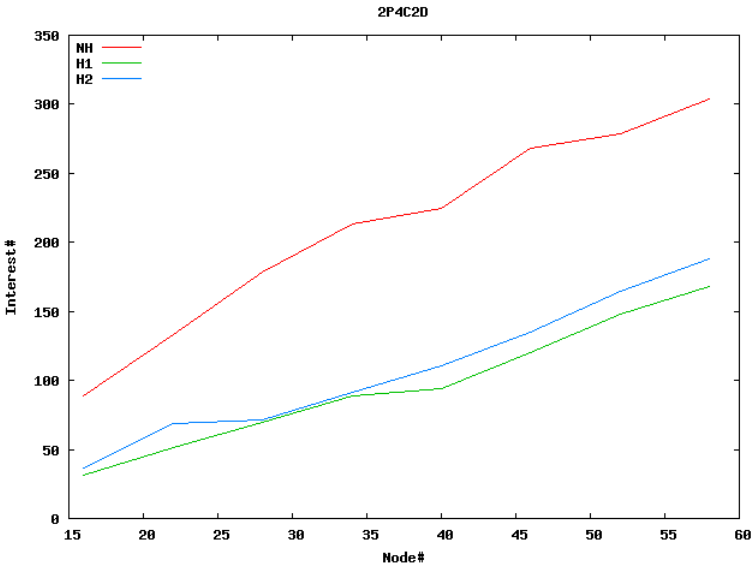


Figure 4.14: *Results - 2P4C2DAnd Nodes vs Interests*

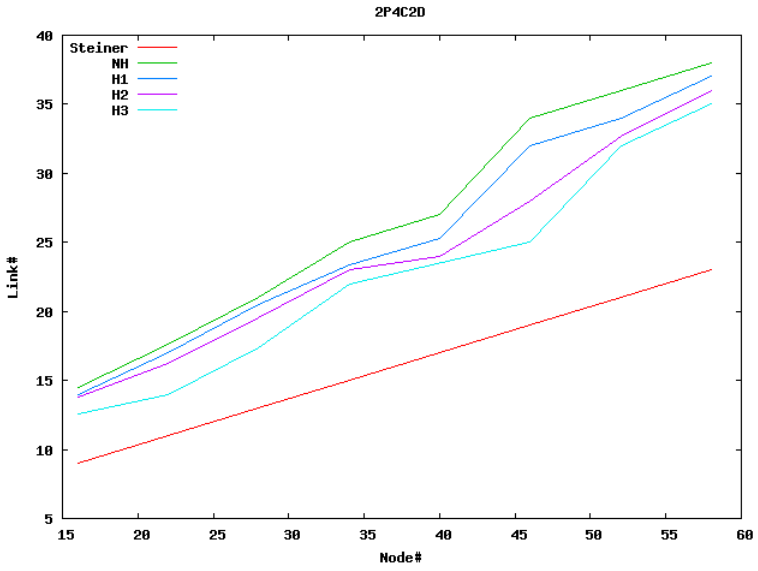


Figure 4.15: *Results - 2P4C2DAnd Nodes vs Links*

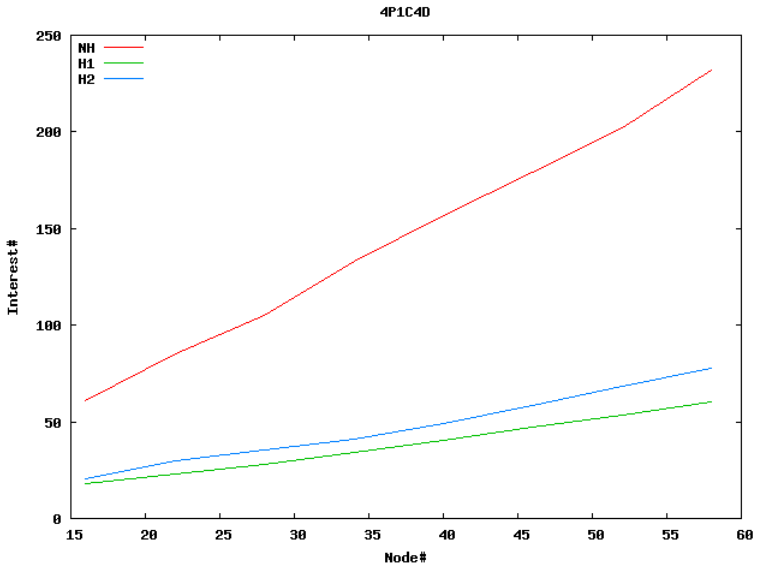


Figure 4.16: *Results - 4P1C4DAnd Nodes vs Interests*

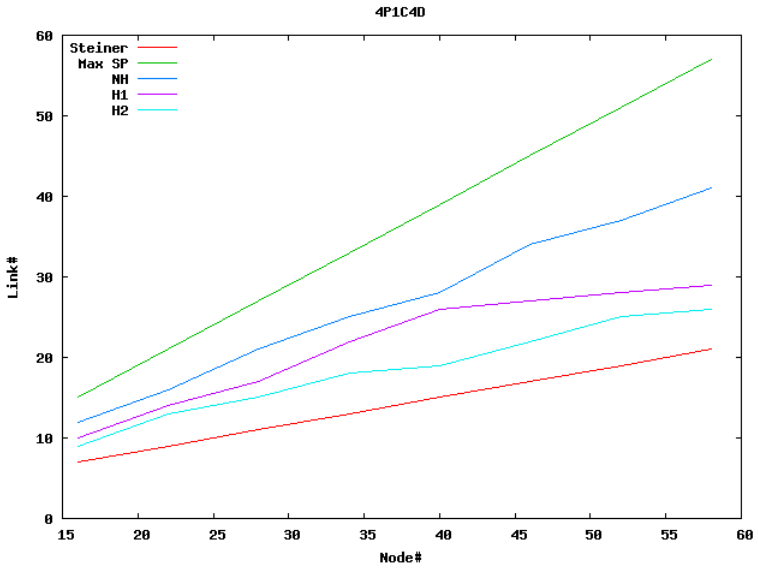


Figure 4.17: *Results - 4P1C4DAnd Nodes vs Links*

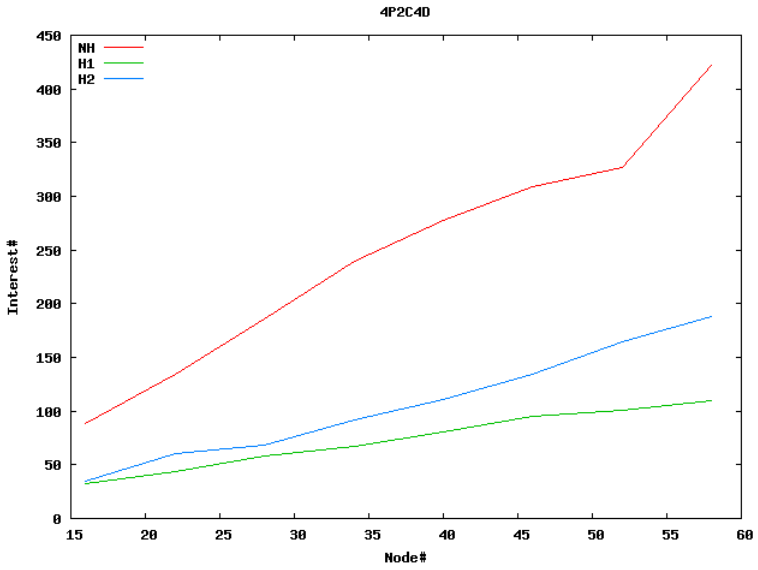


Figure 4.18: *Results - 4P2C4DAnd Nodes vs Interests*

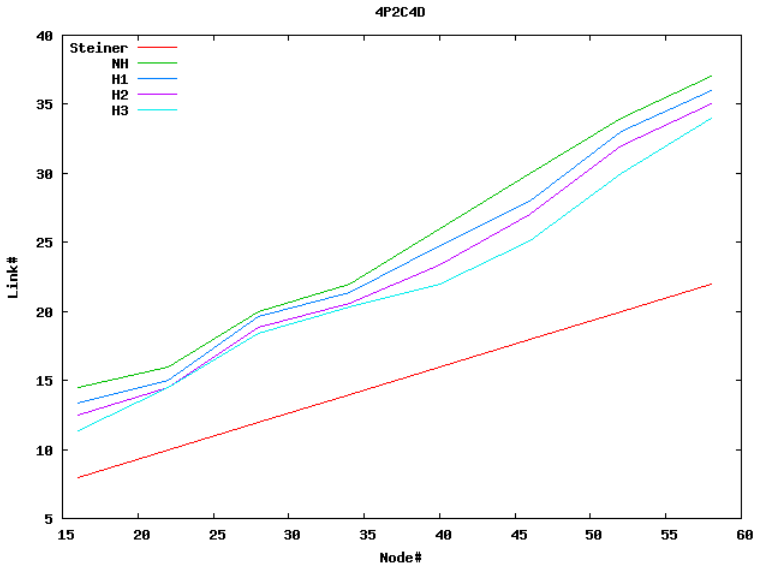


Figure 4.19: *Results - 4P2C4DAnd Nodes vs Links*

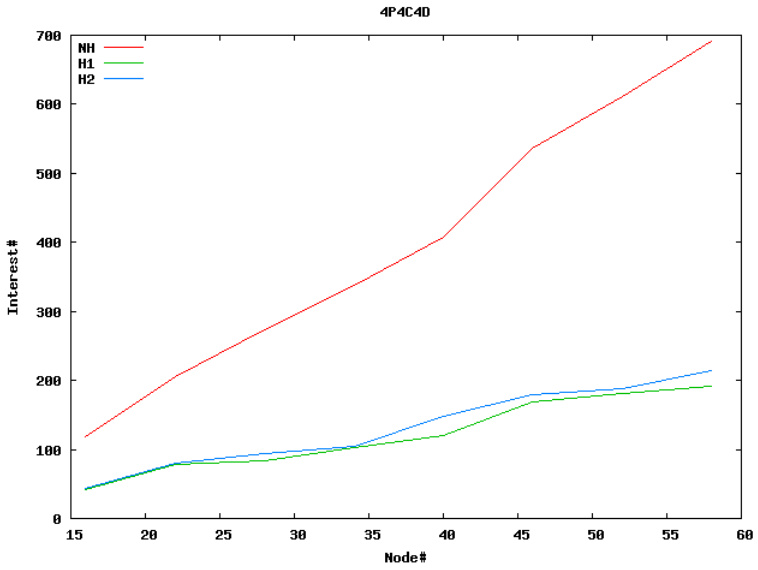


Figure 4.20: *Results - 4P4C4DAnd Nodes vs Interests*

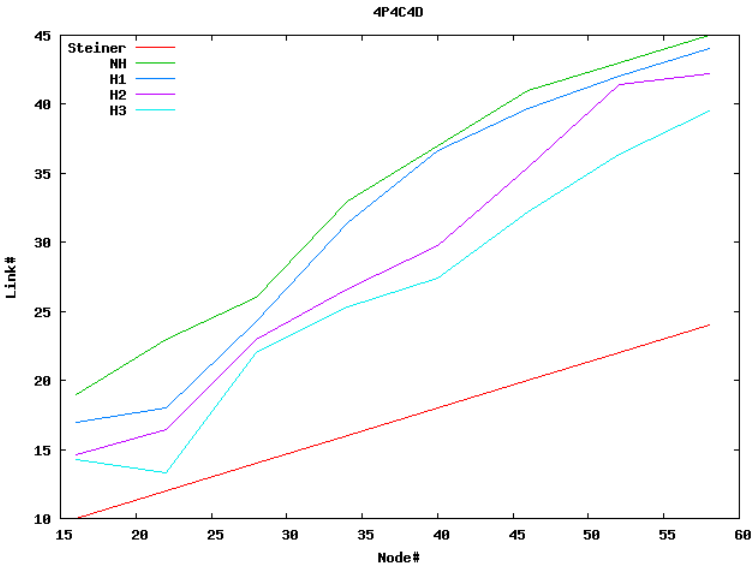


Figure 4.21: *Results - 4P4C4DAnd Nodes vs Links*

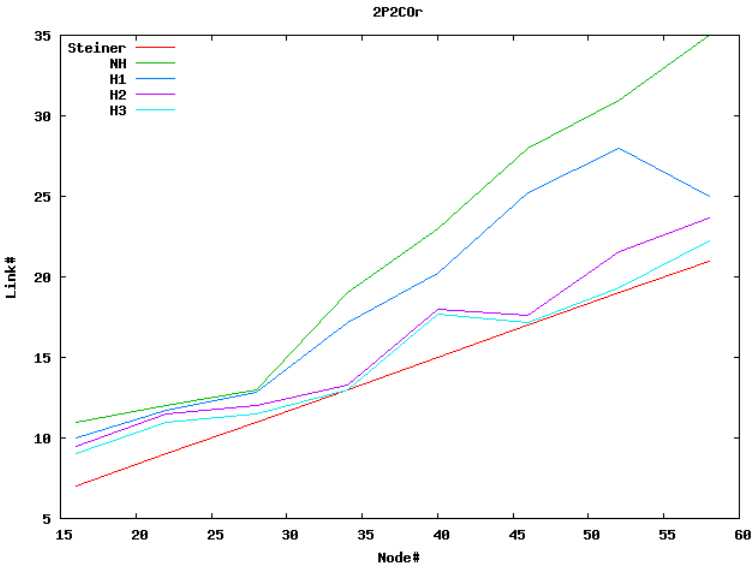


Figure 4.22: *Results - 2P2COr Nodes vs Links*

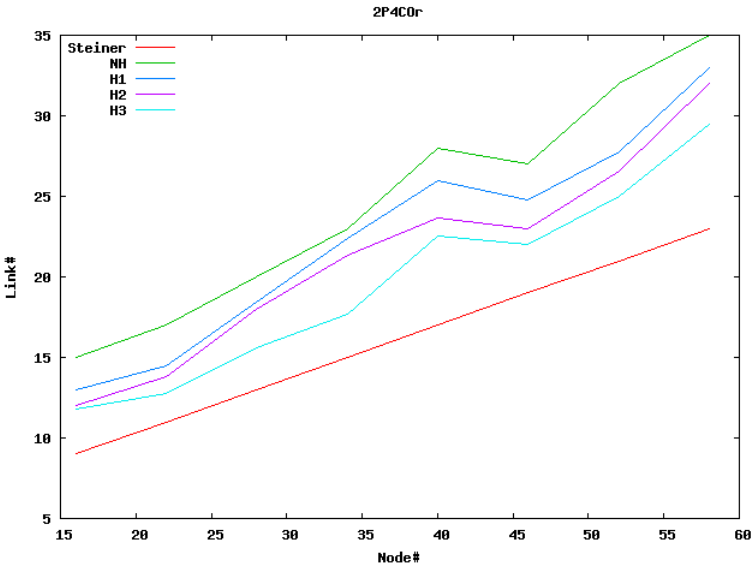


Figure 4.23: *Results - 2P4COr Nodes vs Links*

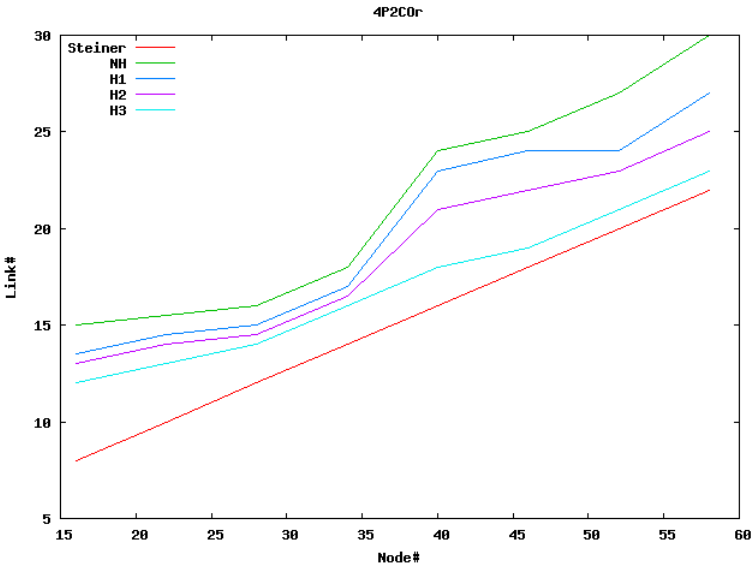


Figure 4.24: *Results - 4P2COr Nodes vs Links*

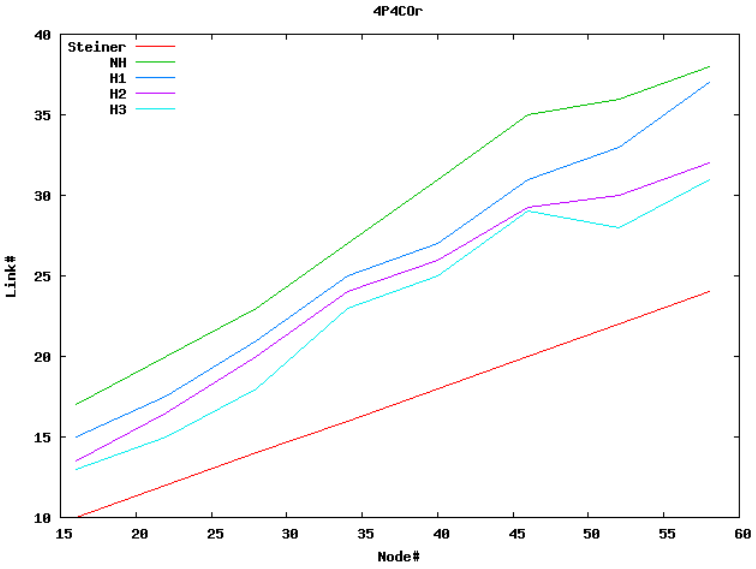


Figure 4.25: *Results - 4P4COr Nodes vs Links*

Chapter 5

Conclusions

From the analysis of results it is clear that use of heuristics can definitely improve the data flow in the network, by selecting shared paths over shorter paths which can lead to more number of transmissions.

1. Aggregation of interest not only lowers the number of interest messages sent out, but also leads to better shared data path graphs where number of data transmissions is lower than the algorithm which does not use any heuristics.

2. The metric used for path selection is a cost metric. Although implemented algorithms use a value proportional to number of hops as cost, it has been observed that implementation of shortest path heuristic together with interest aggregation can lead to lower number of transmissions. Besides, preventing the further propagation of interest when data is already available at a relay lowers the number of interests propagated, if shortest path heuristic alone were implemented.

3. Although cost division heuristic further improves on the other heuristics, as explained in section 4.2.3, the preference for preventing duplicate data paths affects the performance of this algorithm.

5.1 Future Work

The next step to do will be to allow sinks to reinforce lower cost paths that arrive later in the refresh cycle and incorporate negative reinforcements to prevent duplicate data paths. This should improve on the data graphs formed by heuristic algorithms discussed in this thesis.

Another area of study that could be further done is to study these algorithms with a different metric for cost based on the underlying radio model. This study could be done with distance as cost, or power consumed for transmission is another choice for metric.

From the experience of adjusting the timers over extensive testing and data collection of the algorithms, it would be good to have an adaptive timer resolution[14] where the timer values are computed from the transmit time of messages and their reply; and progressively reducing the timer frequency.

Bibliography

[1] V. G. Murphy A.L., Roman G.C., Proceedings of Ninth International Workshop on Software Specification and Design , 25 (1998).

[2] G. Aggelou, *Mobile Ad hoc Networks*, Mc-Graw Hill, 2005.

[3] Research:wireless ad hoc sensor networks, national institute of standards and technology, http://www.antd.nist.gov/wahn_ssn.shtml.

[4] C.-Y. Chong and S. P. Kumar, Proceedings of the IEEE **91**, 1247 (2003).

[5] Y. S. Ian F Akayildiz, Weilian Su and E. Cayirci, IEEE Communications (2002).

[6] B. Krishnamachari, D. Estrin, and S. Wicker, Modelling data-centric routing in wireless sensor networks, In IEEE INFOCOM.

[7] Ravindra.K.Ahuja, Thomas.L.Magnanti, and James.B.Orlin, *Network flows: Theory, Algorithms and Applications*, Prentice Hall, 1993.

[8] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, IEEE/ACM Trans. Netw. **11**, 2 (2003).

[9] B. Krishnamachari and J. Heidemann, Application-specific modelling of information routing in sensor networks, 2004.

[10] J. Hill et al., System architecture directions for networked sensors, in *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[11] D. Gay et al., The nesc language: A holistic approach to networked embedded systems, in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming*

language design and implementation, pages 1–11, New York, NY, USA, 2003, ACM Press.

[12] P. Levis, N. Lee, M. Welsh, and D. Culler, Tossim: accurate and scalable simulation of entire tinyos applications, in *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003, ACM Press.

[13] P. Levis and N. Lee, Tossim: A simulator for tinyos networks, 2003.

[14] J. Thomsen and D. Husemann, Evaluating the use of motes and tinyos for a mobile sensor platform, in *PDCN'06: Proceedings of the 24th IASTED international conference on Parallel and distributed computing and networks*, pages 95–100, Anaheim, CA, USA, 2006, ACTA Press.

Appendix A

Component declaration of Application

Configuration module for the simulator application implementing algorithm with no heuristics.

```
configuration Algorithm {

}

implementation {
  components  Main, TimerC, MoteInfoM, InterestM,
        RouteTableM, IntTableM, GenMsgSend, DataTableM, TrcvMsg,
        DataM, DataMsgSend, DemoSensorC as Sensor;

  Main.StdControl -> MoteInfoM.StdControl; //reads the src/sink info
  Main.StdControl -> TrcvMsg.StdControl;

  Main.StdControl -> InterestM.StdControl;
  Main.StdControl -> GenMsgSend.StdControl;

  InterestM.RouteTableI -> RouteTableM; //keeps track of neighbors
```

```
  InterestM.IntTableI -> IntTableM;
  InterestM.DataTableI -> DataTableM;
  InterestM.GenMsgOutput -> GenMsgSend;
  InterestM.ReinforceTimer -> TimerC.Timer[unique("Timer")];
  InterestM.InterestTimer -> TimerC.Timer[unique("Timer")];
  InterestM.IdInterest -> IdNeighborM;
  InterestM.ReceiveGenMsg -> TrcvMsg;
  InterestM.InterestData -> DataM;
  InterestM.Send -> TrcvMsg;

  Main.StdControl -> DataM.StdControl;
  Main.StdControl -> DataMsgSend.StdControl;

  DataM.RouteTableI ->RouteTableM;
  DataM.IntTableI -> IntTableM;
  DataM.DataTableI -> DataTableM;
  DataM.DataOutput -> DataMsgSend;
  DataM.ReceiveDataMsg -> TrcvMsg;
  DataM.DataInterest-> InterestM;
  DataM.DataTimer -> TimerC.Timer[unique("Timer")];
  DataM.ADC -> Sensor;

  IntTableM.CounterTimer -> TimerC.Timer[unique("Timer")];
}
```

Appendix B

Test Results

Node	NH	H2
16	15	18
22	21	23
28	27	29.8
34	33	41.2
40	39	47.75
46	45	52.33
52	51	57
58	57	69

Table B.1: *1P1C1DAnd - Interest*

Node	Steiner	NH	H2
16	5	5.6	5.4
22	7	7.5	7.25
28	9	9.75	9.4
34	11	12	11.75
40	13	15	13.8
46	15	16.6	16.25
52	17	19	18.75
58	19	20.75	20.5

Table B.2: *1P1C1DAnd - Links*

Node	NH	H1	H2
16	24.2	20.6	26.75
22	35	31	38
28	47.25	42.6	49.33
34	51.5	51.8	59.6
40	64.25	61	69.4
46	73.25	72.6	85
52	90	83.6	96.8
58	94	86.5	122

Table B.3: *1P2C1DAnd - Interest*

Node	Steiner	NH	H1	H2	H3
16	6	9.33	8.8	8	7
22	8	11.8	11	10	9.4
28	10	14	13	12.25	12.1
34	12	16.67	15	13.67	13.33
40	14	20	17	16	15
46	16	23	20	18	16.67
52	18	26.5	22	20.67	20
58	20	29.5	23.2	21.75	21.67

Table B.4: *1P2C1DAnd - Link*

Node	NH	H1	H2
16	39	27.2	27.2
22	72.25	40	42
28	96	56	56.2
34	127	75.4	87.4
40	143	86.33	97.8
46	167	92.25	117.2
52	190	103	123.67
58	215	125	130.4

Table B.5: *1P4C1DAnd - Interest*

Node	Steiner	NH	H1	H2	H3
16	8	14	11.67	10.25	10.2
22	10	16	13.8	12.25	12
28	12	18	16.33	16.18	15.6
34	14	20	18.4	18.13	17.33
40	16	21.33	20.8	20.5	19.9
46	18	24	23.2	22.22	20.9
52	20	27.5	26.75	26.5	25.67
58	22	30	27.75	27	26

Table B.6: *1P4C1DAnd - Link*

Node	NH	H1	H2
16	30	16	19.4
22	42	22	24.33
28	54	28	35.8
34	66	34	45
40	78	40	47
46	90	46	70.25
52	102	52	73
58	114	58	105

Table B.7: *2P1C2DAnd - Interest*

Node	Steiner	NH	H1	H2
16	6	9.8	8.13	8
22	8	13.5	11.75	9
28	10	15.5	14.5	10.75
34	12	21.8	15.33	14.5
40	14	22.57	18.25	16.67
46	16	26.25	19	17.6
52	18	28.2	20.67	19.5
58	20	31	23.5	23.25

Table B.8: *2P1C2DAnd - Link*

Node	NH	H1	H2
16	56	27.2	28.4
22	85	34.4	41
28	102	46.4	55
34	128	52.5	69.17
40	150	69.2	92.8
46	163	80.4	103
52	190	97.6	117.25
58	212	101.4	149.67

Table B.9: *2P2C2DAnd - Interest*

Node	Steiner	NH	H1	H2	H3
16	7	13	11.67	11	9.86
22	9	16	12.33	11.7	10.67
28	11	19.67	14.75	13.4	13
34	13	20.5	19.33	19	17.5
40	15	26.5	24.6	22	21.67
46	17	28	26	23.25	23
52	19	33	32	28.5	28.33
58	21	36	32.4	29.25	28

Table B.10: *2P2C2DAnd - Link*

Node	NH	H1	H2
16	89	31.25	36.25
22	133.33	51.2	69
28	178.5	70	71.75
34	213	88.8	91.8
40	224.5	94.33	110.4
46	268	120	134.8
52	279	147.6	164.6
58	304	168.25	188.25

Table B.11: *2P4C2DAnd - Interest*

Node	Steiner	NH	H1	H2	H3
16	9	14.5	14	13.75	12.6
22	11	17.6	17	16.25	14
28	13	21	20.5	19.5	17.4
34	15	25	23.33	23	22
40	17	29	26	25.25	24
46	19	34	32	28	25
52	21	36	34	32.67	32
58	23	39	37	36	35

Table B.12: *2P4C2DAnd - Link*

Node	NH	H1	H2
16	61	17.75	20.25
22	85	23.2	29.8
28	105	28.2	35.67
34	133	34	41.25
40	157	40.2	49
46	179	47.5	58.4
52	202	53.75	68.67
58	232	60.5	78

Table B.13: *4P1C4DAnd - Interest*

Node	Steiner	NH	H1	H2
16	7	12	10	9
22	9	16	14	13
28	11	19.8	17	15
34	13	23	22	18
40	15	27.7	26	19
46	17	30	27	22
52	19	34	28	25
58	21	38	29	26

Table B.14: *4P1C4DAnd - Link*

Node	NH	H1	H2
16	88.75	32.2	35.2
22	134.67	43.4	61
28	186	57.8	68.2
34	239	67.25	91.8
40	278	80.4	110.4
46	308.67	95	134.8
52	326.33	101	164.6
58	421.67	110	188.25

Table B.15: *4P2C4DAnd - Interest*

Node	Steiner	NH	H1	H2	H3
16	8	14.5	13.33	12.5	11.33
22	10	16	15	14.5	14.5
28	12	21	19.67	18.8	18.4
34	14	25.1	21.33	20.6	20.33
40	16	28.2	24.8	23.33	22
46	18	32.2	28	27	25.14
52	20	36.7	33	32	30
58	22	40	36	35	34

Table B.16: *4P2C4DAnd - Link*

Node	NH	H1	H2
16	117.6	44	42.6
22	206	79.6	78.4
28	273	83.33	94.2
34	337	104.17	102.6
40	407	120.33	148.2
46	535.67	169.6	179.8
52	610.5	181	188
58	691.67	191.25	213.8

Table B.17: *4P4C4DAnd - Interest*

Node	Steiner	NH	H1	H2	H3
16	10	19	17	14.6	14.25
22	12	23	18	16.4	13.33
28	14	26	24.25	23	22
34	16	33	31.4	26.6	25.33
40	18	37	36.67	29.8	27.4
46	20	41	39.67	35.4	32.2
52	22	43	42	41.4	36.4
58	24	45	44	42.2	39.5

Table B.18: *4P4C4DAnd - Link*

Node	Steiner	NH	H1	H2	H3
16	7	11	10	9.5	9
22	9	12	11.75	11.5	11
28	11	13	12.8	11.5	11.5
34	13	19	17.2	13.25	13
40	15	23	20.25	18	17.67
46	17	28	25.25	17.6	17.2
52	19	31	28	21.6	19.33
58	21	35	25	23.67	22.25

Table B.19: *2P2C2DOr - Link*

Node	Steiner	NH	H1	H2	H3
16	9	15	13	12	11.8
22	11	17	14.5	13.8	12.75
28	13	20	18.4	18	15.6
34	15	23	22.4	21.33	17.67
40	17	28	26	23.67	22.5
46	19	27	24.75	23	22
52	21	32	27.75	26.6	25
58	23	35	33	32	29.5

Table B.20: *2P4C2DOr - Link*

Node	Steiner	NH	H1	H2	H3
16	8	15	13.5	13	12
22	10	15.5	14.5	14	13
28	12	16	15	14.5	14
34	14	18	17	16.5	16
40	16	24	23	21	18
46	18	25	24	22	19
52	20	27	24	23	21
58	22	30	27	25	23

Table B.21: *4P2C4DOr - Link*

Node	Steiner	NH	H1	H2	H3
16	10	17	15	13.5	13
22	12	20	17.5	16.5	15
28	14	23	21	20	18
34	16	27	25	24	23
40	18	31	27	26	25
46	20	35	31	29.25	29
52	22	36	33	30	28
58	24	38	37	32	31

Table B.22: *4P4C4DOr - Link*