SIMULATION DESIGN AND ANALYSIS OF ENERGY-EFFICIENT DATA REDISTRIBUTION IN SELFISH BASE STATION-LESS SENSOR NETWORKS

A Thesis

Presented

to the Faculty of

California State University Dominguez Hills

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Andre Chen

Fall 2018

ACKNOWLEDGEMENTS

First and foremost, I thank my mother, Wenju Chen, and my father, Dienyih Chen, for all of the support and guidance they have given me throughout my life. I also express my gratitude to my advisor and Committee Chair, Dr. Bin Tang, as well as committee members Dr. Mohsen Beheshti and Dr. Jianchao Han, for all of their feedback and advice during the development of this thesis. And finally, a special thanks to my mentors at The Aerospace Corporation, Richard Johnson and Supannika Mobasser, for their unwavering support of my personal and professional development.

TABLE OF CONTENTS

LIST OF TABLES

PAGE

LIST OF FIGURES

PAGE

ABSTRACT

Today's data-driven society has prompted the need for energy-efficient data collection via data-intensive sensor networks (DISNs). Previous research assumed that the nodes in these networks willingly cooperate with each other when routing and storing data. This is unrealistic because there is no guarantee that nodes work together. A more realistic network consists of nodes that are selfish, which act only in their own self interest regardless of the consequences to other nodes. In 2016, Chen and Tang showed that selfish nodes can be incentivized to behave in such a way that energy consumption is minimized and all data is stored (Chen, Tang, 2016). This thesis extends their work by illustrating how to design a simulation that empirically verifies their results and how to modify the simulation parameters to model all 21 cases required for successful verification. It also briefly describes how the simulation helped refine their theory.

CHAPTER 1

INTRODUCTION

Data-intensive wireless sensor networks are used to collect large quantities of data in areas where it is not feasible to set up a base station, such as near earthquake faults, under the ocean, and in areas hit by natural disasters (Tang, Jaggi, Wu, Kurkal, 2013). Sensors deployed in this area have limited battery power and storage, which introduces two primary problems. First, their limited battery power means that they can only send and receive a limited amount of data before they are no longer able to operate. Second, their limited storage means that there is only a finite amount of data that each sensor can store before it must relay additional offloaded data to other nodes, thus draining precious battery power.

The two problems of limited battery power and limited storage capacity, combined with the absence of a base station to replenish power or collect data, makes data collection difficult. In order to preserve all data collected, nodes must store as much data as they can, and then offload the rest to other nodes, which can quickly deplete their battery power. Furthermore, the nodes do not necessarily cooperate willingly with each other: they may be owned by separate entities whose goals might conflict with energy-efficient data collection. Because cooperation cannot be guaranteed, and because nodes may behave in the best interest of themselves rather than the group, they must be incentivized to participate in such a way as to benefit the group while still achieving their

own goals. It turns out that there is a way to incentivize all nodes to participate such that all data is preserved and energy consumption is minimized (Chen, Tang 2016).

This thesis presents both a simulation design and relevant simulation data that empirically verifies the results from (Chen, Tang 2016). Its main goal is to explicitly show concrete examples of how a base station-less wireless sensor network behaves under the conditions of limited battery power, limited storage capacity, and selfish behavior.

Problem Description

This section discusses the key concepts of the energy-efficient data redistribution problem. These concepts provide the theoretical foundation for the simulation that empirically verifies the theoretical results in (Chen, Tang, 2016).

A data-intensive sensor network can be modeled by a graph, $G = (V, E)$ where V is a set of vertices and E is a set of weighted, directed edges. Each vertex represents an individual wireless sensor node. Each edge is defined to be a pair of nodes, (n_1, n_2) that are both within transmission range of each other. That is, they are close enough such that they can send and receive data from each other. The nodes are partitioned into two groups: *data generator* nodes and *data storage* nodes. The data generators receive data from the outside environment, but must offload this data to other nodes (storage nodes). The storage nodes receive data and either store it or transmit it to other nodes.

Data received by data generators is sent to a sequence of nodes until it reaches a designated storage node. This sequence of nodes is known as a *data preservation path*

and is determined by the Minimum Cost Flow (MCF) algorithm (Goldberg, 1997). The MCF algorithm identifies a set of energy-efficient data preservation paths based on a collection of source-destination pairs, the storage constraints on all nodes in the network, and the total cost to route one data item along the path.

When data is routed throughout the network, the total energy cost of the routing is based on cost parameters inherent to each node. Every node has three cost parameters: the cost to transmit a bit on a circuit (ε_{elec}), the cost to transmit a bit on a transmit amplifier (ε_{amp}), and the cost to store one bit (ε_{store}). These cost parameters, whose true values are known only to the nodes themselves, affect three different actions that nodes can take: receiving data, transmitting data, and storing data. The cost to receive data is based on the cost to transmit on a circuit and is given by: $b * \varepsilon_{elec}$ where b is the number of bits to receive. The cost to send data from one node n_{source} (the source) to an adjacent node n_{target} (the target) is based on the costs to transmit along circuit and transmit amplifier, and the distance between the source and destination node. It is given by:

$$b * \varepsilon_{amp} * dist\left(n_{source}, n_{target}\right)^2 + b * \varepsilon_{elec}$$

The expression $dist\left(n_{source}, n_{target}\right)^2$ represents the square of the distance between the source and target node. The cost to store data is based only on the storage parameter and is given by $b * \varepsilon_{store}$. The goal of energy-efficient data preservation is to guarantee that all data is stored in storage nodes and the total amount of energy consumed by all nodes is minimized.

It is possible to accomplish the goal of energy-efficient data preservation under the assumption that all nodes cooperate by instructing all nodes to follow the appropriate

algorithm (Tang, Jaggi, Wu, Kurkal, 2013). However, if the nodes are selfish and will not willingly cooperate with each other, the algorithm in (Tang, Jaggi, Wu, Kurkal, 2013) will not naturally occur without some additional constraints. These additional constraints manifest themselves in the form of a mechanism, a concept from game theory. In game theory, a mechanism defines a set of strategies, or decisions, for each agent (each selfish node, in this case). The nodes in a data-intensive sensor network can either tell the truth or lie about a particular cost parameter. The choice that each node makes is known as that node's *strategy*. Every node has a specific amount of utility it will gain based on a payment function. The payment to each node is given by:

$$p_i(\tilde{c}_i, c_{-i}) = c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i)$$

And the utility for each node is given by:

$$\pi_i(\tilde{c}_i, c_{-i}) = p_i - c_i = c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i$$

The symbols in the payment and utility equations are explained in Table 1.

Table 1. Explanation of symbols in payment and utility equations.

Symbol	Meaning
$\tilde{c}_i$	The sum of all costs that node *i* incurs based on its reported costs.
c_{-i}	The strategies of all nodes other than node *i*
$c_{V-\{i\}}$	The minimum total cost required to route all data when node *i* does not participate
$\tilde{c}_V$	The minimum total cost required to route all data when node *i* participates
c_i	The sum of all costs that node *i* incurs based on its true costs
$p_i(\tilde{c}_i, c_{-i})$	The payment owed to node *i* based on its reported costs $\tilde{c}_i$ and the strategies of all nodes other than node *i* (c_{-i})
$\pi_i(\tilde{c}_i, c_{-i})$	The utility of node *i* based on its reported costs $\tilde{c}_i$ and the strategies of all nodes other than node *i* (c_{-i})

Due to the assumption that each node is selfish, rather than cooperative, each node is interested in maximizing only its own utility, regardless of what happens to the overall network. By cleverly constructing the payment function and mechanism, and by routing all data in via the minimum cost flow algorithm, it is possible to incentivize all nodes to participate in data preservation in such a way that all data is preserved (no data loss) and total energy consumption is minimized (energy is used most efficiently) (Chen, Tang, 2016).

The cleverly constructed mechanism in (Chen, Tang, 2016) belongs to a family of mechanisms known as *Vickrey-Clark-Groves (VCG) mechanisms* where the dominant strategy is to always tell the truth about one's private types. That is, no node is incentivized to lie about its costs because its utility is maximized only under truth-telling. Mechanisms in the VCG family are known as *truthful mechanisms* and guarantee that all agents will truthfully report their private types, thus revealing the hidden information intrinsic to each private type. This makes analysis of these networks easier because all true information is made available to outsiders.

Data preservation proceeds in three stages. The first stage, the *report stage* is when all nodes report their private types. Note that every node has the option to lie about the value of a cost parameter. The second stage is the *compute cost stage* where all costs are computed based on the reported types. Since the nodes have the option to lie about their costs, this reported cost is not necessarily the true cost. Note that the nodes themselves are aware of their own true cost parameters, so they also compute a true cost for themselves, which is used when computing payment and utility. After costs are

computed, the MCF algorithm is run based on the costs and underlying network graph to determine the data paths along which data must be routed to ensure all data is preserved and energy consumption is minimized. The third and final stage, the *decision stage* occurs when all nodes make decisions to either participate or not participate in the data preservation based on their individual strategies. If the mechanism and payment functions are all defined in the way specified in (Chen, Tang, 2016), then at the end of the game, all data has been preserved and the minimum amount of energy has been consumed.

Major Assumptions

There are several major assumptions that are crucial to the solution of the data preservation problem. First, the underlying network must be biconnected. That is, the graph is connected (there always exists at least one path from one node to another node) and the removal of any one node from the graph does not affect connectivity (there is still at least one path from one node to another). The assumption of biconnectivity is reasonable: networks that are designed to collected data are built with redundancy in mind and one way to do this is by ensuring that the network graph is biconnected.

The second assumption is that the underlying network is *feasible*. The network is considered *feasible* if all nodes will always have enough battery power such that they will always be present in network. The study of the infeasible case is beyond the scope of this work and is left for future research.

The third assumption is that all nodes are aware of the rules of the game and know how to compute their payment and utility functions based on the rules. This assumption is reasonable because each node in the network is informed beforehand how it will be paid. Without this knowledge, nodes would not know how to compute their payments, and thus would not know how to compute their utilities, which they want to maximize. If this knowledge were not available, no nodes would participate at all since there is no guarantee of any payment in such a system.

The fourth assumption is that all nodes are selfish. This is reasonable because it more closely matches the real world where individual agents have their own goals that they seek to achieve, rather than only achieving the goals of the group (non-selfishness, full cooperation). A more complex variant of the data preservation problem relaxes the assumption of pure selfishness such that some nodes will collude with other nodes to maximize utility for a subset of nodes rather than an individual node. This collusion problem is more difficult to analyze and is left for future research.

The fifth assumption is that there is enough storage capacity in the network such that the removal of any single storage node still leaves enough capacity to store all data items. This assumption is reasonable because networks are usually built to be redundant: the loss of a node or set of nodes should have minimal effect on the total network capacity. Otherwise, the network is too fragile to be utilized (too many single points of failure, or too many possibilities for significantly reduced network capacity). Note that without this assumption, the computation of utilities cannot be performed because they

require the computation of minimum cost flow for the underlying graph where one node and its incident edges are removed.

The sixth assumption is that data generator nodes are already incentivized to participate, and thus do not need additional motivation to offload data. This is a reasonable assumption because they can be motivated beforehand to participate in receiving data from the environment and offloading it to nodes. Note that storage nodes do not have this flexibility because there is no guarantee that they will be chosen to participate in data routing, and thus cannot demand prior payment for their service.

Problem Significance

The energy efficient data preservation problem is significant for two reasons. First, as the world becomes more interconnected, more and more data flows throughout various networks that are essential to critical infrastructure. Understanding how nodes in these networks behave under different circumstances helps network participants, researchers, businesses, and many other stakeholders achieve their goals. The nodes in the networks do not necessarily cooperate with each other, and so understanding how to incentivize these nodes to cooperate to achieve goals for the common good can assist business leaders, researchers, and policymakers when they are involved in the development and implementation of these networks. Second, understanding how to simulate and analyze these networks, as well as verify that they are behaving in accordance with the rules, is crucial when designing and implementing complex networks. Simulation analysis and verification contributes to society's overall

understanding of network behavior and how it can be utilized for the public good, all without requiring expensive infrastructure implementations that may waste resources or actively harm the public. Without simulation and modeling, it is much riskier to implement these networks due to the relatively high levels of uncertainty.

CHAPTER 2

LITERATURE REVIEW

This section discusses literature relevant to the theoretical solution presented in (Chen, Tang, 2016), followed by a discussion of how a simulation supports theoretical results.

There are several examples in the current literature that show how algorithmic mechanism design is applicable to data routing, network flow, and data redistribution problems. The term *Algorithmic Mechanism Design* was introduced by Noam Nisan and Amir Ronen in their seminal paper on the topic (Nisan, Ronen, 1999). In their paper, they present a solution to a task-scheduling problem, which they solve by designing an algorithm that utilizes payments to achieve a desired outcome. This combination of algorithm and payment specification is described as a *mechanism*, a concept from game theory, and serves as the foundation for several solutions to problems related to the energy-efficient data redistribution problem.

In (Feigenbaum, Papadimitriou, Sami, Shenker, 2005) the authors present a mechanism that incentivizes agents to truthfully report their costs and to willingly route traffic along Least Cost Paths (LCPs), which minimizes the total energy consumed when routing data. However, they assume that all agents in the network have sufficient storage capacity, so their mechanism is not directly applicable to data redistribution in wireless networks whose nodes have finite storage. In (Anderegg, Eidenbenz, 2003), a routing protocol called *Ad-hoc VCG* is presented, which is guaranteed to find cost-efficient paths in mobile (wireless) networks where all nodes are incentivized to truthfully report their

costs. However, as in the previous example, it is assumed that all nodes have sufficient storage to receive data, so the protocol cannot be used to redistribute data in storage-constrained wireless networks.

In all the aforementioned examples, the correctness of results is shown via mathematical proofs. However, in some cases, it is useful to analyze simulations of real-world scenarios to verify that the results are consistent with the theory. One example of this technique was used to show that particular quantity was always within theoretical upper and lower bounds (Anderegg, Eidenbenz, 2003). A different example, which uses simulation to help design systems rather than to verify results, is the simulation framework for industrial wireless systems presented in (Liu, Candell, Lee, and Moayeri, 2016). Although the primary purpose of the simulation in this thesis is to verify results, it might also help in the design of data-intensive sensor networks by providing relevant cost data for designers interested in predicting operating costs.

The results in (Chen, Tang, 2016), which form the theoretical foundation for this thesis, are shown to be correct via mathematical proof. However, two elements are missing that can provide additional insight into the behavior of wireless sensor networks. First, there is no simulation presented that accurately models the network and the behavior of its nodes, so it is unclear how a real world network would behave when the nodes operate under the mechanism presented in (Chen, Tang, 2016). Second, the paper identifies all cases in which the utility of each node is maximized, but does not provide examples of this behavior in a network. This thesis proposes to bridge these two gaps in

the literature by presenting both a simulation that models the wireless network and results that verify all theoretical cases presented in (Chen, Tang, 2016)

CHAPTER 3

METHODOLOGY

This section describes the design of a simulation that models a data-intensive wireless sensor network and how this simulation is used to empirically verify theoretical results in (Chen, Tang, 2016).

Simulation Design

The purpose of the simulation is to empirically verify the results in (Chen, Tang 2016) by computing the utility of storage nodes when they lie about their costs and comparing it to the utility when they tell the truth. The results are considered verified if and only if the utility is maximized under the truth-telling strategy.

The following is a list of components that the simulation requires when computing utilities:

1. Vertex object – Represents a node in the network. Each vertex has a unique identifier, cost parameters, unique 2D position, transmission range, and helper methods (accessors, mutators, equality and hashing methods, etc.)
2. Edge object – Represents a pair of nodes in the network that can communicate with each other. It contains a unique identifier, the pair of nodes defining the

edge, and the cost to transmit one bit from the source node (tail) to the destination node (head).

3. Graph object – Represents the network. Contains information about its underlying set of vertices and edges, and contains helper functions (internal tests of connectivity and biconnectivity, node removal operations, accessor and mutators for properties, etc.)
4. Dijkstra Algorithm Object – A special object that invokes an instance of Dijkstra's algorithm on a graph. There are two inputs to the algorithm: a graph that represents the underlying network, and a fixed specified node in the graph. The output is a collection of shortest paths from the specified node to all other nodes in the graph.

The interaction between components is based on the steps in (Chen, Tang, 2016) that are required when computing utilities. The simulation performs the following steps to compute the utilities:

1. Generate Nodes - Generate a set of nodes based either on a user-supplied file or an algorithm that randomly generates a set of nodes.
2. Construct Edges - Based on the transmission range of the nodes provided by the user, construct all pairs of nodes that are within transmission range.
3. Construct Graph - Based on the nodes and edges constructed, construct a graph.

4. Verify Graph Properties - Test the graph for biconnectedness. If it is biconnected, proceed. If not, randomly generate nodes and edges until a biconnected graph is constructed.
5. Compute MCF Input File Under Truth-telling - Based on the biconnected graph, generate an input file that summarizes the graph and node information so that the MCF algorithm can properly read the information and produce the appropriate output. Assume all nodes tell the truth about their costs.
6. Compute Preservation Paths - Run the MCF algorithm on the input file to generate a set of paths along which data will be routed. The algorithm also computes the total energy required to route all the data. This is the MCF cost under truth-telling when all nodes participate.
7. Compute Costs with a Node Removed - For each storage node, construct a modified version of the underlying biconnected graph by removing a specific storage node and repeat the steps needed to compute the MCF cost. This will yield the MCF cost when a specific node is removed.
8. Compute Utilities Under Truth-telling - Compute the utility of each storage node under the truth-telling strategy based on the payment and utility functions in (Chen, Tang, 2016). The utility is given by $\pi_i(\tilde{c}_i, c_{-i}) = c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i$, with symbols defined in Table 1.
    a. Compute $c_{V-\{i\}}$, the MCF cost when node *i* is removed from the network. This is done by modifying the graph. Node *i* is removed and all of its

incident edges are removed. The MCF algorithm is executed on this modified graph and the cost computed by the algorithm is $c_{V-\{i\}}$

b. Compute $\tilde{c}_V$, the MCF cost when node *i* is included in the network. This is done by running the MCF algorithm on the graph where all nodes follow a truth-telling strategy (they report the true costs of all of their parameters). Note that in this case, the reported costs of all nodes is equal to the true cost of all nodes because the utility under a truth-telling strategy is being computed.

c. Compute $\tilde{c}_i$, the cost that Node *i* incurs based on its reported costs. In this case, since a truth-telling strategy is used, this is equivalent to computing the cost that Node *i* incurs based on its true costs. The program will examine all data paths where Node *i* is a participant and add up all the costs that Node *i* incurs in these cases.

d. Compute c_i, the cost that Node *i* incurs based on its true costs. This computation is done the exact same way as the computation for $\tilde{c}_i$ in the previous step because all nodes are using a truth-telling strategy. Note that this is not the case when a non-truth-telling strategy is employed.

e. Compute $c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i$ based on the previous computations. The result is the utility under truth-telling for Node *i*.

9. Construct Biconnected Graphs Under a Non-truthful Strategy - For each storage node, construct a modified version of the underlying biconnected graph. This time, the storage node lies about its costs based on user-supplied parameters.

10. Compute Utilities Under a Non-truthful Strategy - Compute the utility of each storage node under a non-truthful strategy based on the payment and utility functions in (Chen, Tang, 2016). The utility is given by $\pi_i(\tilde{c}_i, c_{-i}) = c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i$, with symbols defined in Table 1. For each node being investigated (one at a time), let that node adjust the cost of exactly one of its cost parameters and scale it by some positive scale factor. Let all other nodes not being investigated use any strategy they wish. Let Node *i* be the node being investigated.
    a. Compute $c_{V-\{i\}}$, the MCF cost when node *i* is removed from the network. This is done by modifying the graph. Node *i* is removed and all of its incident edges are removed. The MCF algorithm is executed on this modified graph and the cost computed by the algorithm is $c_{V-\{i\}}$. Note that this is exactly the same number that was computed in the utility computation under a truth-telling strategy. The reason is because when a node is removed from a network, its strategy has no effect on the costs of any nodes because it is not present in the network at all, and hence cannot use different strategies to affect costs.
    b. Compute $\tilde{c}_V$, the MCF cost when node *i* is included in the network. This is done by running the MCF algorithm on the graph where Node *i* adopts a non-truthful strategy by lying about one of its cost parameter values. Note that the underlying graph in this case is different from the graph in the truthful strategy case because the MCF algorithm is only aware of

reported costs. This is why a different graph had to be constructed when analyzing the non-truthful case.

c. Compute $\tilde{c}_i$, the cost that Node *i* incurs based on its reported costs. The program will examine all data paths where Node *i* is a participant and add up all the costs that Node *i* incurs in these cases, based on the reported cost parameters rather than the true cost parameters.

d. Compute c_i, the cost that Node *i* incurs based on its true costs. This computation is done the exact same way as the computation for $\tilde{c}_i$ in the previous step, except all costs are based on the true cost parameters rather than the reported cost parameters.

e. Compute $c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i$ based on the previous computations. The result is the utility under a non-truthful strategy for Node *i*.

11. Collect Results - Insert the utility of each storage node under different strategies (truth-telling vs. lying) into a data structure. Continue insertion and computations until all storage nodes have been examined.
12. Verify Results - Verify that utility is maximized for each node only under the truth-telling strategy. Account for precision errors caused by representation of real numbers in a finite number of bits.
13. Generate Summary - Generate a summary file with all utilities of all storage nodes in a common data format, such as a comma-separated-value (CSV) file.

Simulation Workflow

This section contains a sequence of steps that illustrate how to set up and run the simulation.

1. Specify a set of simulation parameters. The list of parameters are as follows:
    a. Network Dimensions – A pair of numbers that represents the physical space in which the network resides. The pair is of the form (X, Y) where X is the upper bound on the x-coordinate of a network node and Y is the upper bound on the y-coordinate of a node. Units for X and Y are in meters. For example, if the pair (400, 700) is provided, then the network resides in a 400 meter x 700 meter grid where the x-coordinate of all nodes is in the interval [0,400] and the y-coordinate of all nodes is in the interval [0,700].
    b. Number of data generators – An integer that represents the number of data generators in the network. This integer must be at least 1.
    c. Number of storage nodes – An integer that represents the number of storage nodes in the network. This integer must be at least 1.
    d. Transmission range – A number that represents the transmission range of every node in the network as measured in meters. It must be greater than 0.
    e. Storage capacity – An integer that represents the total number of data items that can be stored by each storage node. It must be greater than or

equal to 0. Note that this does not apply to data generators, which have no storage capacity.

f. True value of cost parameters for all nodes.
    i. True value of ε_{elec}– a number that represents the true number of joules required to transmit one bit on a node's circuit. Must be greater than or equal to 0.
    ii. True value of ε_{amp}– a number that represents the true number of joules required to transmit one bit on a node's transmit amplifier. Must be greater than or equal to 0.
    iii. True value of ε_{store} – a number that represents the true number of joules required to store one bit. Must be greater than or equal to 0.

g. Reported value of cost parameters for all nodes
    i. Reported value of ε_{elec} – a number that represents the reported number of joules required to transmit one bit on a node's circuit. Must be greater than or equal to 0.
    ii. Reported value of ε_{amp} – a number that represents the reported number of joules required to transmit one bit on a node's transmit amplifier. Must be greater than or equal to 0.
    iii. Reported value of ε_{store} – a number that represents the reported number of joules required to store one bit. Must be greater than or equal to 0.

h. Number of data items – an integer that represents the total number of data items that that will preserved in the network. The integer must be greater than or equal to 0.

i. Number of bytes per data item – an integer that represents the size of one data item, as measured in bytes.

j. Filepath for node summary file – an absolute filepath that points to the location of a node summary file, which includes information about node coordinates and unique node identifiers. This filepath is optional – the simulation will generate this node summary file if it is not present.

2. Run the simulation and provide the configuration file as input.

3. The simulation will create a sub-directory within its main directory (where the simulation executable is stored) based on the current time. All simulation logs and results will be stored in this sub-directory.

    a. Summary of utilities – a CSV file containing all unique node identifiers, their true utilities, and their reported utilizes, will be produced when the simulation finishes all of its computations.

    b. Log files – A text file that lists all of the main steps that the simulation performs. Its purpose is to aid in debugging as well as provide additional detail about the nodes and the underlying network.

    c. MCF input files – A collection of text files generated that serve as input to the MCF algorithm under various conditions. The general input to the MCF program is a list of source-destination pairs where each pair also has

an upper and lower bound on capacity (based on the capacities of the storage nodes) as well as a number that represents the total amount of energy consumed when routing from the source to the destination.

d. MCF output files – A collection of text files that describe the data preservation paths that were chosen by the MCF algorithm. Each file contains a collection of source-destination pairs, the number of data items that were routed from each source to each destination, and the total amount of energy consumed to route and store all data.

Data Analysis

This section describes how to analyze the data generated by the simulation. The CSV file that the simulation generates is a summary of all the true and reported utilities for a specific simulation run. The data in the CSV file and the path information from the simulation logs can be used to verify that the results match the theory in (Chen, Tang, 2016). To conduct this verification, fix any one specific node and check the following cases enumerated in (Chen, Tang, 2016):

1. The node can lie about its ε_{store} parameter. All other parameters are truthfully reported. This is Theorem 1 in (Chen, Tang, 2016).
    a. The node is not selected to participate regardless of whether it tells the truth or lies. Utility should be exactly 0 in both cases.

b. The node is selected to participate when it tells the truth, but is not selected to participate when it lies. Utility under truth-telling should be greater than or equal to utility under lying.
c. The node is not selected to participate when it tells the truth, but is selected when it lies. The utility under truth-telling should be 0 while the utility under lying should be less than or equal to zero.
d. The node is in the preservation path when it tells the truth as well as when it lies. Four subcases must be checked.
    i. The node is instructed to store data when it tells the truth and when it lies. Its utility should be exactly equal in both cases.
    ii. The node is instructed to store data when it tells the truth, but is instructed to relay data to another node if it lies. The utility under truth-telling should be greater than or equal to the utility under lying.
    iii. The node relays data when it tells the truth, but stores data when it lies. The utility under truth-telling should be greater than or equal to the utility under lying.
    iv. The node relays data regardless of whether it tells the truth or lies. The utilities should be exactly the same.

2. The node can lie about its ε_{amp} parameter. All other parameters are truthfully reported. This is Theorem 2 in (Chen, Tang, 2016).

a. The node is not selected to participate regardless of whether it tells the truth or lies. Utility should be exactly 0 in both cases.
b. The node is selected to participate when it tells the truth, but is not selected to participate when it lies. Utility under truth-telling should be greater than or equal to utility under lying.
c. The node is not selected to participate when it tells the truth, but is selected when it lies. The utility under truth-telling should be 0 while the utility under lying should be less than or equal to zero.
d. The node is in the preservation path regardless of whether it tells the truth or lies. There are four sub-cases to check.
    i. The node is instructed to store data when it tells the truth and when it lies. Its utility should be exactly equal in both cases.
    ii. The node is instructed to store data when it tells the truth, but is instructed to relay data to another node if it lies. The utility under truth-telling should be greater than or equal to the utility under lying.
    iii. The node relays data when it tells the truth, but stores data when it lies. The utility under truth-telling should be greater than or equal to the utility under lying.
    iv. The node relays data when it tells the truth and when it lies. Its utility under truth-telling should be greater than or equal to the utility when it lies.

3. The node can lie about its ε_{elec} parameter. All other parameters are truthfully reported. This is Theorem 3 in (Chen, Tang, 2016).
    a. The node is not selected to participate regardless of whether it tells the truth or lies. Utility should be exactly 0 in both cases.
    b. The node is selected to participate when it tells the truth, but is not selected to participate when it lies. Utility under truth-telling should be greater than or equal to utility under lying.
    c. The node is not selected to participate when it tells the truth, but is selected when it lies. The utility under truth-telling should be 0 while the utility under lying should be less than or equal to zero.
    d. The node is in the preservation path regardless of whether it tells the truth or lies. There are four sub-cases to check.
        i. The node is instructed to store data when it tells the truth and when it lies. Its utility should be exactly equal in both cases.
        ii. The node is instructed to store data when it tells the truth, but is instructed to relay data to another node if it lies. The utility under truth-telling should be greater than or equal to the utility under lying.
        iii. The node relays data when it tells the truth, but stores data when it lies. The utility under truth-telling should be greater than or equal to the utility under lying.

iv. The node relays data when it tells the truth and when it lies. Its utility under truth-telling should be greater than or equal to the utility when it lies.

In all of the aforementioned cases, the utility under truth-telling and the utility under lying is compared, and the results of these comparisons are verified against the theory in (Chen, Tang, 2016). However, an additional step is required to account for the fact that real numbers are encoded in a finite number of bits, which produces precision error when computing utilities. For example, in one simulation run, Node 40 lies about its ε_{amp} parameter and reports that it is 10% of its true value. Its true utility is computed to be 960.6547133220884 and its reported utility is 960.6547133220882, a difference of about 1.1368683772161603E-13. This difference is 15 orders of magnitude off of the utility values, is 16 orders of magnitude off of the parameters ε_{elec} and ε_{store}, and is 19 orders of magnitude off the ε_{amp} parameter. This suggests that the difference may be due to a precision error rather than a computation error. To verify this, the simulation computes the difference in magnitudes, which can be used to compare against some threshold for precision error.

After all cases have been checked and any very small errors in numbers have been identified as precision errors (and thus, not contradictory of theory), the simulation run has empirically verified the theoretical results for a particular set of parameters. The simulation can be run multiple times with different parameters to verify the results.

CHAPTER 4

RESULTS

This section describes the results of a simulation run to show how the simulation data was used to empirically verify the results in (Chen, Tang, 2016).

Graph Illustration

This section provides a visual depiction of the graph for a specific simulation run. The simulation parameters for this example are known as the *half-full homogeneous case*, which means that each node has the same true cost parameters and the total incoming data is half of the total network capacity. The parameters are as follows:

1. Network Size = 1000 meters * 1000 meters
2. Transmission Range = 250 meters
3. Number of nodes = 50
4. Number of data generators = 10
5. Number of storage nodes = 40
6. Default ε_{elec} = 100 nanojoules = 100 * 10^{-9} Joules
7. Default ε_{amp} = 100 picojoules = 100 * 10^{-12} Joules
8. Default ε_{store} = 100 nanojoules = 100 * 10^{-9} Joules
9. Default storage capacity = 50 data items
10. Number of data items per data generator = 100 data items
11. Number of bits per data item = 512 Bytes = 512 * 8 bits = 4096 bits

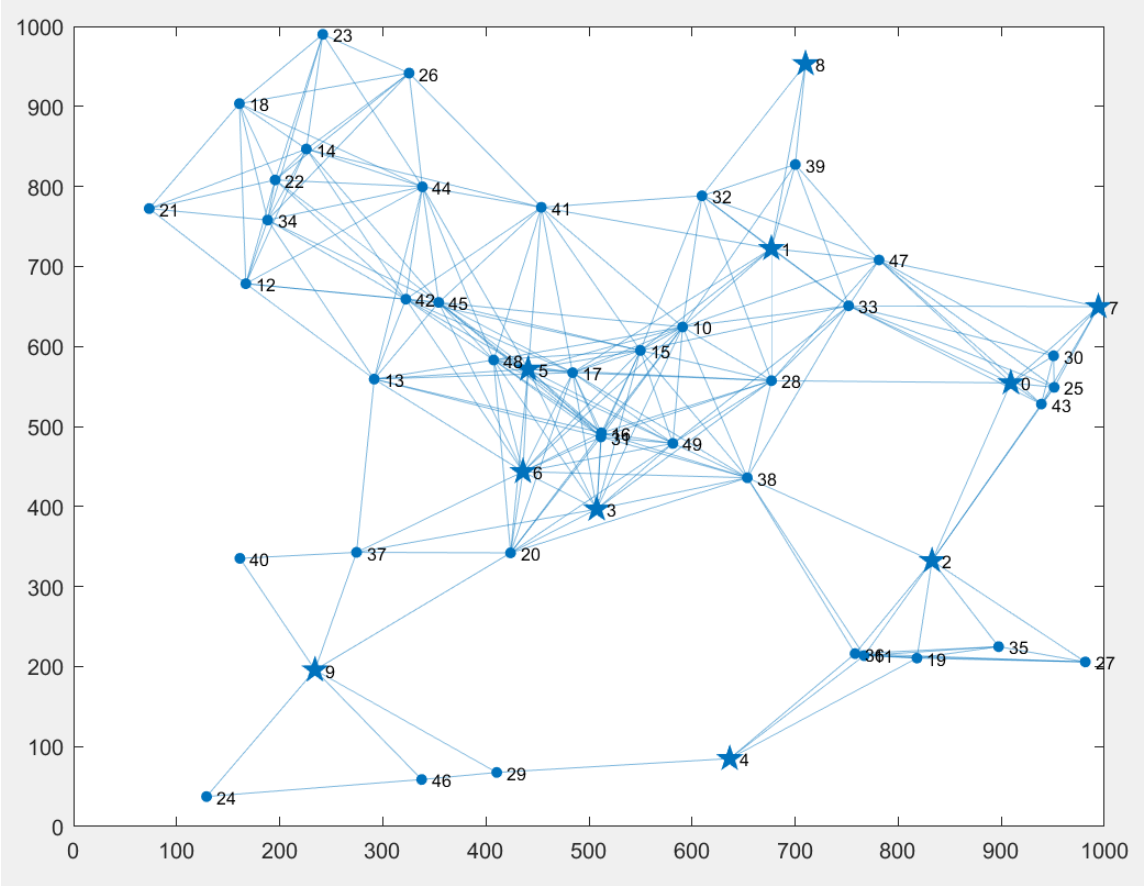


Figure 1. Graph representation of underlying network

A list-based description of the image is as follows:

- Each node is identified by a unique integer, 0-49 inclusive.
- Nodes 0-9 are data generators
- Nodes 10-49 are data storage nodes
- Each node has a unique position in 2-dimensional space
- A line joining two nodes is an edge, which indicates that both source and destination nodes are within transmission range of each other and can thus communicate (send and receive data) with each other.

- The graph is a directed graph where an edge (N1,N2) is a directed edge from node N1 (source) to node N2 (destination). The directed edge (N1, N2) means that data moves from node N1 to node N2. Note that (N1, N2) and (N2, N1) represent different edges with possibly different total costs: N1 and N2 can have different cost parameters, so even though the distance between the nodes is the same, cost to transmit from N1 to N2 can be different from the cost to transmit from N2 to N1. To keep the image uncluttered, there is only one line between any two nodes within transmission range. Thus, the line joining nodes N1 and N2 represents both (N1, N2) and (N2, N1). This is only for visualization purposes; the simulation treats the edges as distinct objects.

Verification of Theoretical Results

This section illustrates how simulation runs can verify the results in (Chen, Tang, 2016). The general technique is as follows: for each of the three cost parameters (ε_{elec}, ε_{amp}, ε_{store}), fix the value of two parameters and allow a node to lie about the third parameter's value. Compute the true utility and reported utility of this node and verify that utility is always maximized under truth-telling in all the cases listed in Theorem 1, Theorem 2, and Theorem 3 in (Chen, Tang, 2016). Do this for all storage nodes in the network. For each of the verifications, the half-full homogeneous case (see APPENDIX A: Node Summary Files) is used as a baseline for the nodes' cost parameters. Deviations from the half-full homogeneous case are noted in each case where they occur.

1. Theorem 1 Verification – Suppose a node lies about its ε_{store} parameter and consider the following cases for a specific node.
    a. Case 1 – The node is not in the data preservation path regardless of its strategy. Its utility is 0 in both cases. This is illustrated by Node 12 when its ε_{store} parameter is scaled by 0.1 Its true and reported utility is 0 in both cases because it is not selected to participate in any data preservation. This confirms Case 1.
    b. Case 2 - The node is selected to participate when it tells the truth, but is not selected to participate when it lies. Utility under truth-telling should be greater than or equal to utility under lying. Node 37 participates under truth-telling and has utility 1272.816519261134, but when ε_{store} is scaled by 10.0, it is now too expensive to store data in this node, so MCF does not select it to participate. Its utility when it reports ε_{store} scaled by 10.0 is 0. This confirms Case 2.
    c. Case 3 - The node is not selected to participate when it tells the truth, but is selected when it lies. The utility under truth-telling should be 0 while the utility under lying should be less than or equal to zero. Modify the original homogeneous case so that the true ε_{amp} parameter for Node 10 is set to 0.01 and ε_{store} is set to 0.1. Under truth-telling, Node 10 is not selected to participate because its storage and relay costs are too high, so its utility is 0. When Node 10 lies about its ε_{store} parameter and scales it by 0.1, the MCF algorithm selects it for participation as a storage node.

Although Node 10 now participates, its utility is negative: -18216.9678643829, which is less than its utility under truth-telling (0). This confirms Case 3.

d. Case 4 – The node is in the preservation path when it tells the truth as well as when it lies. Four subcases must be checked.
    i. Subcase 4-1 – The node stores data under both truth-telling and lying strategies. The utility is expected to be equal in both cases. Node 43 has utility 9999.419662404369 when it tells the truth as well as when it lies by scaling ε_{store} by 0.1. This confirms Subcase 4-1.
    ii. Subcase 4-2 - The node is instructed to store data when it tells the truth, but is instructed to relay data to another node if it lies. The utility under truth-telling should be greater than or equal to the utility under lying. Node 10 is a storage node under truth-telling (preservation path {1,10}), but is a relay node when it reports ε_{store} is scaled by 10.0 (preservation path {1,10,15}). Under truth-telling, Node 10's utility is 2058.2321356170723, but when it lies, its utility is 1129.124814741197. Thus, Node 10's utility is maximized under truth-telling. This confirms Subcase 4-2.
    iii. Subcase 4-3 - The node relays data when it tells the truth, but stores data when it lies. The utility under truth-telling should be greater than or equal to the utility under lying. Set Node 10's

ε_{store} parameter to 0.1. Leave all other parameters the same as in the homogeneous case. Under truth-telling, Node 10 participates in routing along path {1,10,15} as a relay node and has utility 1129.124814741197. When Node 10 lies and reports ε_{store} scaled by 0.01, it routes along path {1,10} as a storage node and has utility -18216.967864382932, which is less than its utility under truth-telling. This confirms Subcase 4-3.

iv. Subcase 4-4 - The node relays data regardless of whether it tells the truth or lies. The utilities should be exactly the same. Node 30 relays data along path {7,30,25} under truth-telling and lying (ε_{store} scaled 0.1). The utility in both cases is the same: 11530.483056185709. This confirms Subcase 4-4.

Figure 2 illustrates the true vs. reported utilities in the half-full homogeneous case when nodes lie about their ε_{store} parameter. It also illustrates how the reported utilities change as ε_{store} is scaled by different factors. For example, Node 17 acquires the same amount of utility when its storage parameter is scaled by 0.01, 0.1, and 1.0 (true cost). However, when it inflates its cost by 10.0 and 100.0, it gains less than half the utility under truth-telling. This shows how a node that tries to inflate its costs in the hopes of gaining more utility will in fact gain less.

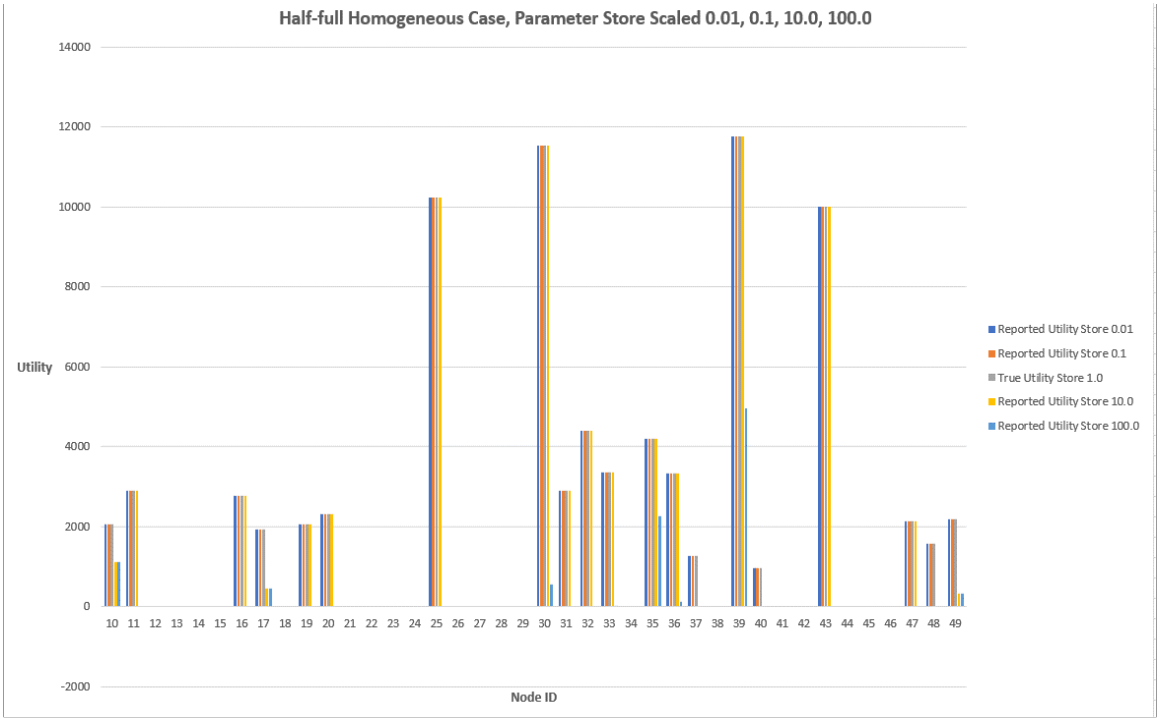


Figure 2. Utilities in the half-full homogeneous case with scaled store parameter

2. Theorem 2 Verification – Let nodes lie about their ε_{amp} parameter and consider the following cases for a specific node.
    a. Case 1 - The node is not selected to participate regardless of whether it tells the truth or lies. Utility should be exactly 0 in both cases. Node 12 is not selected when it tells the truth, and when it lies (ε_{amp} scale 0.1), it is still not selected to participate. Its utility is 0 in both cases. This confirms Case 1.
    b. Case 2 - The node is selected to participate when it tells the truth, but is not selected to participate when it lies. Utility under truth-telling should be greater than or equal to utility under lying. Set Node 10's true

parameters to (ε_{elec}, ε_{amp}, ε_{store}) = (0.001, 0.000001, 0.1), which is the homogeneous case except ε_{store} is 0.1 instead of 0.001. Under truth-telling, Node 10 acts as a relay node along {1,10,15} and has utility 1129.124814741197. When it lies about ε_{amp} and reports ε_{amp} = 0.01, it is no longer selected to participate and so its utility is 0. This confirms Case 2.

c. Case 3 - The node is not selected to participate when it tells the truth, but is selected when it lies. The utility under truth-telling should be 0 while the utility under lying should be less than or equal to zero. Set Node 10's true parameters to (ε_{elec}, ε_{amp}, ε_{store}) = (0.001, 0.01, 0.1) and scale ε_{amp} by 0.0001 so that the reported parameter vector is (0.001, 0.000001, 0.1). Under truth-telling, Node 10 does not participate and its utility is 0. Under lying where ε_{amp} is scaled by 0.0001, Node 10's utility is negative and is given by -5193424.576623198, so it is less than utility under truth-telling. This confirms Case 3.

d. Case 4 - The node is in the preservation path regardless of whether it tells the truth or lies. There are four sub-cases to check.

    i. Subcase 4-1 - The node is instructed to store data when it tells the truth and when it lies. Its utility should be exactly equal in both cases. Under the homogeneous case parameters, Node 10 stores data under truth-telling and also stores data under lying with ε_{amp}

scaled by 0.1. The utility in both cases is 2058.2321356170723. This confirms Subcase 4-1.

ii. Subcase 4-2 - The node is instructed to store data when it tells the truth, but is instructed to relay data to another node if it lies. The utility under truth-telling should be greater than or equal to the utility under lying. Set Node 10's true parameters to (ε_{elec}, ε_{amp}, ε_{store}) = (0.0001, 0.001, 0.001). Set Node 15's true parameters to (ε_{elec}, ε_{amp}, ε_{store}) = (0.0001, 0.000001, 0.0001). Let all other nodes' true parameters be the same in the homogeneous case. Then under truth-telling, Node 10 has utility 2242.552135617065 and is routed along {1,10} where it acts only as a storage node. When Node 10 lies and scales ε_{amp} by 0.0001, it is routed along {1,10,15} as a relay node, and has utility -517121.4087402644. Thus, Node 10 still maximizes utility under truth-telling, and under lying, it has negative utility. This confirms Subcase 4-2.

iii. Subcase 4-3 - The node relays data when it tells the truth, but stores data when it lies. The utility under truth-telling should be greater than or equal to the utility under lying. Set Node 10's true parameters to (ε_{elec}, ε_{amp}, ε_{store}) = (0.0001, 0.0000001, 0.001) and set Node 15's true parameters to be (ε_{elec}, ε_{amp}, ε_{store}) = (0.0001, 0.000001, 0.0001). When Node 10 tells the truth, it relays data along path {1,10,15} and has utility 2333.96140352948.

When Node 10 lies and scales ε_{amp} by scale factor 10000, it stores data and has utility 2242.552135617065, which is less than the utility under truth-telling. This confirms Subcase 4-3.

iv. Subcase 4-4 - The node relays data when it tells the truth and when it lies. Its utility under truth-telling should be greater than or equal to the utility when it lies. Use the same parameters that were used in Subcase 4-3: Set Node 10's true parameters to (ε_{elec}, ε_{amp}, ε_{store}) = (0.0001, 0.0000001, 0.001) and set Node 15's true parameters to be (ε_{elec}, ε_{amp}, ε_{store}) = (0.0001, 0.000001, 0.0001). When Node 10 tells the truth, it relays data along path {1,10,15} and has utility 2333.96140352948. When it lies and scales ε_{amp} by 0.5, it has the same utility: 2333.96140352948. This confirms Subcase 4-4.

Figure 3 provides a summary of true utilities vs. reported utilities when ε_{amp} is scaled by various factors. It also illustrates how the reported utilities change when the ε_{amp} parameter is changed. For example, Node 16 experiences negative utility when it scales its amp parameter by 0.01, and gains less than utility under truth-telling when scaling it by 0.1. It gains no advantage when it increases the parameter by 10.0 or 100.0.

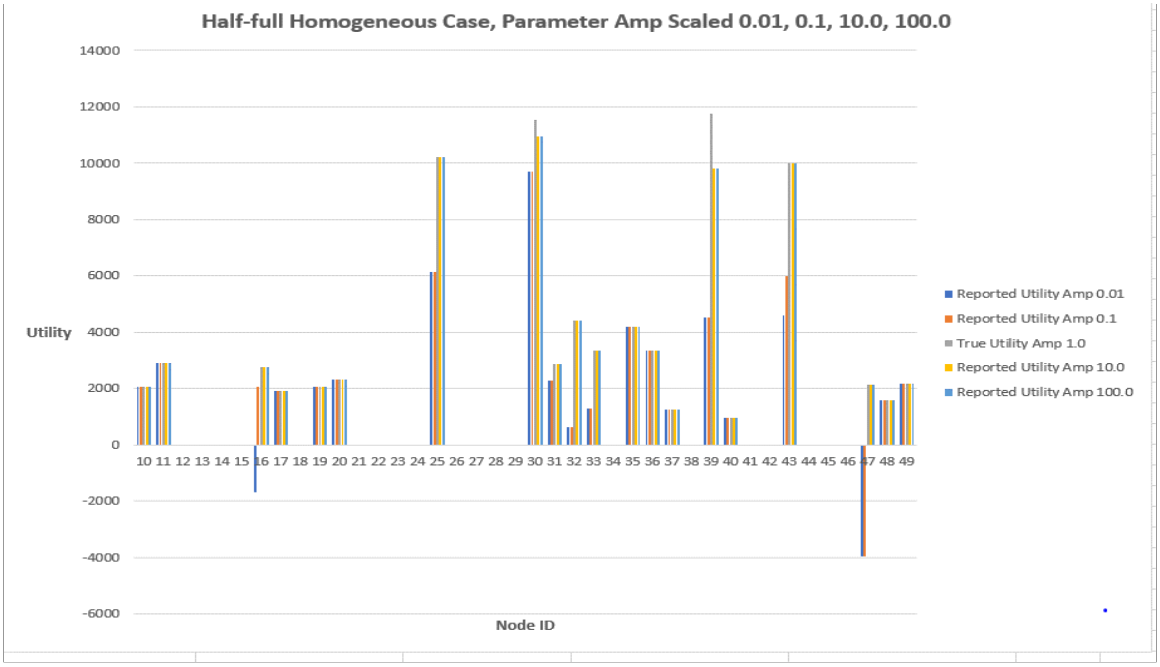


Figure 3. Utilities in the half-full homogeneous case with scaled amp parameter

3. Theorem 3 Verification – Suppose a node lies about its ε_{elec} parameter and consider the following cases for a specific node.
    a. Case 1 - The node is not selected to participate regardless of whether it tells the truth or lies. Utility should be exactly 0 in both cases. Node 12 in the homogeneous case does not participate under truth-telling and also does not participate when it scales its ε_{elec} parameter by 0.1 Utility is 0 is both cases. This confirms Case 1.
    b. Case 2 - The node is selected to participate when it tells the truth, but is not selected to participate when it lies. Utility under truth-telling should be greater than or equal to utility under lying. Node 37 in the homogeneous case participates under truth-telling and has utility

1272.816519261134. It does not participate when scaling ε_{elec} by 10.0 and has utility 0. This confirms Case 2.

c. Case 3 - The node is not selected to participate when it tells the truth, but is selected when it lies. The utility under truth-telling should be 0 while the utility under lying should be less than or equal to zero. Set all nodes' parameters to the homogeneous case except for Node 10. Set Node 10's true ε_{elec} parameter to 1.0. Node 10 is not selected to participate in this case and has utility 0. When Node 10 lies about its ε_{elec} parameter scaled by 0.001, it is selected to participate, but has negative utility, computed to be -202536.96786438292, which is less than utility under truth-telling. This confirms Case 3.

d. Case 4 - The node is in the preservation path regardless of whether it tells the truth or lies. There are four sub-cases to check.

    i. Subcase 4-1 - The node is instructed to store data when it tells the truth and when it lies. Its utility should be exactly equal in both cases. Node 10 stores data when it tells the truth in the homogeneous case, and also stores data when it lies about its ε_{elec} parameter by scaling it by 0.1. Its utility in both cases is 2058.2321356170723. This confirms Subcase 4-1.

    ii. Subcase 4-2 - The node is instructed to store data when it tells the truth, but is instructed to relay data to another node if it lies. The utility under truth-telling should be greater than or equal to the

utility under lying. In the homogeneous case, Node 31 is instructed to store data when it tells the truth. It has utility 2889.2334227411193. When it lies and scales ε_{elec} by 0.1, it is selected to both store some data and relay the rest. Its utility under lying is 2588.2214664051344, which is less than the utility under truth-telling. Note that Node 31 has two roles when it lies and routes twice as much data as when it tells the truth, but its utility is still maximized under truth-telling. This confirms Subcase 4-2

iii. Subcase 4-3 - The node relays data when it tells the truth, but stores data when it lies. The utility under truth-telling should be greater than or equal to the utility under lying. In the homogeneous case, Node 39 both relays data and stores data under truth-telling, and has utility 11774.11498785732. When Node 39 lies about its ε_{elec} parameter and scales it by 10.0, it only stores data and has utility 9800.370128119832. Its utility is still maximized under truth-telling. This confirms Subcase 4-3.

iv. Subcase 4-4 - The node relays data when it tells the truth and when it lies. Its utility under truth-telling should be greater than or equal to the utility when it lies. In the homogeneous case, Node 30 stores and relays along paths {7,30} and {7,30,25} when it tells the truth and when it lies about its ε_{elec} parameter, scaled by 10.0. The utility under truth-telling is 11530.483056185709 and the

utility under lying is 10961.04455698209. Thus, even though Node 30 has the exact same roles under truth-telling and lying, and routes the same amount of data along the same paths, its utility is still maximized when it tells the truth rather than when it lies. This confirms Subcase 4-4.

Figure 4 summarizes true vs. reported utilities in the half-full homogeneous case when nodes lie about their ε_{elec} parameters. It also illustrates how the reported utilities change based on how ε_{elec} is reported. In this particular example, there are many cases where there is no difference between truth-telling vs. lying. However, the data for Node 30 shows that if it inflates its ε_{elec} parameter by 10.0, it gains less utility than under truth-telling. And if it inflates its ε_{elec} parameter by 100.0, it gains 0 utility because it becomes too expensive to utilize for data preservation.

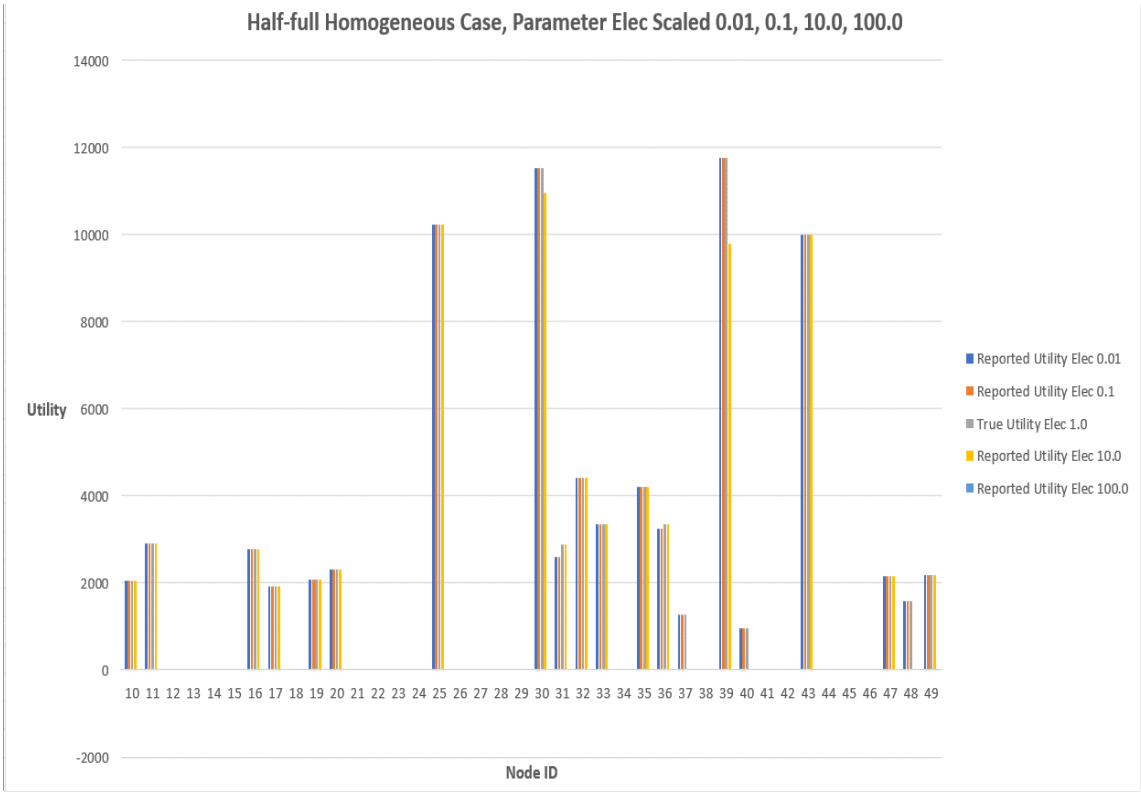


Figure 4. Utilities in the half-full homogeneous case with scaled elec parameter

To provide contrast with the half-full homogeneous case, several other simulation runs were performed to illustrate three other cases: full homogeneous, half-full heterogeneous, and full heterogeneous.

Note that if the amount of incoming data exactly matches the storage capacity of the network, then no analysis is necessary: all data must be stored in all storage nodes regardless of costs.

Full Homogeneous Case

In the full homogeneous case, each node has the same cost parameters and all storage nodes except one are filled to capacity. One storage node is unused because the computation of utilities requires that the MCF cost be calculated for a network where one node is removed from the graph. If all storage nodes were filled to capacity prior to removal of a node, then the network would not have enough capacity to store all data when a node is removed for the calculation of utilities.

Figure 5 shows the utilities in the full homogeneous case when nodes lie about their amp parameter and how the utilities are affected when they scale their amp parameter by different factors. For example, Node 10 has negative utility when its amp parameter is scaled by 0.01 and 0.1, which shows that Node 10 gains significant disadvantage if it suggests that its cost are significantly less than they actually are. When it increases its amp parameter by 10.0 or by 100.0, it gains less utility than under truth-telling, showing that truth-telling is its best strategy to maximize its utility.

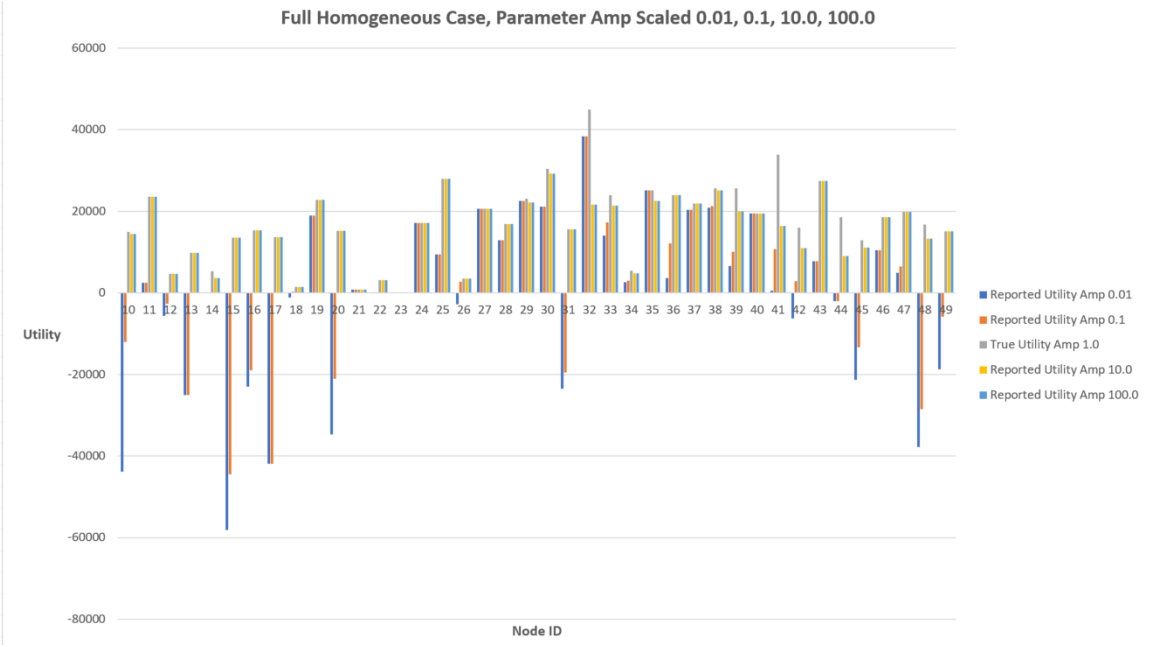


Figure 5. Utilities in the full homogeneous case with scaled amp parameter

Figure 6 shows the utilities in the full homogeneous case when nodes lie about their elec parameter and how the utilities are affected when they are scaled by different factors. For example, Node 32 gains the same amount of utility when it scales its elec parameter by 0.01, 0.1, and 1.0 (true cost), but acquires less utility when it inflates the cost by 10.0 or 100.0. This shows that the payment and utility functions punish Node 32 when it tries to suggest that its costs are larger than they actually are.

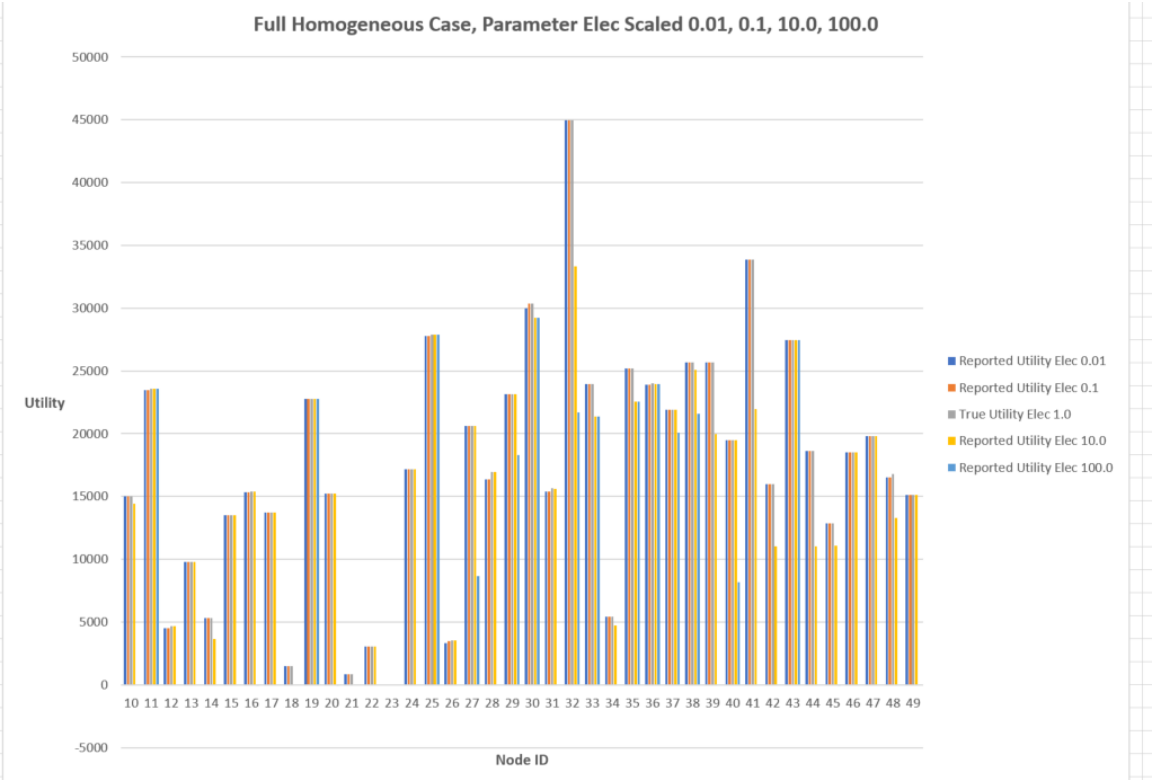


Figure 6. Utilities in the full homogeneous case with scaled elec parameter

Figure 7 shows the utilities in the full homogeneous case when nodes lie about their store parameter and how the utilities are affected when they are scaled by different factors. For example, Node 34 gains the same amount of utility when its store parameter is scaled by 0.01, 0.1, 1.0 (true cost), and 10.0, but less utility when it scales it by 100.0. This shows that in some cases, lying by a certain amount does not affect the acquired utility, but if the lie is significant enough, it will negatively affect the amount of utility gained. One way to think of this is to consider the hypothetical extreme case: imagine that the true cost of a parameter is finite, but a node decides to lie and say its cost is infinite. If all other cost parameters of other nodes are finite, then this node will never be chosen to participate because it would be too costly to allow its participation.

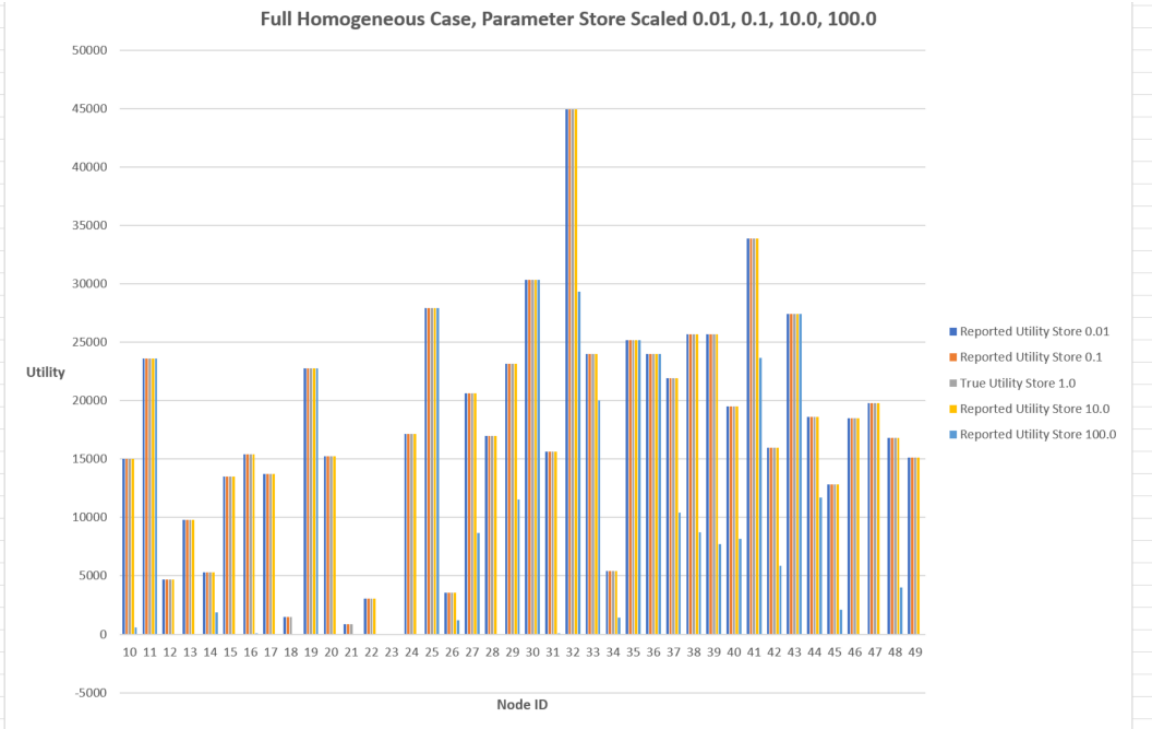


Figure 7. Utilities in the full homogeneous case with scaled store parameter

Half-Full Heterogeneous Case

In the heterogeneous half-full case, each node has randomized cost parameters and the total incoming data is half of the total network capacity. Figure 8 shows the utilities in the half-full heterogeneous case when nodes lie about their elec parameter and how the utilities are affected when they are scaled by different factors. For example, Node 31 gains maximum utility under truth-telling, but gains less utility when it scales its elec parameter by 0.01 and 0.1, and zero utility when it scales it by 10.0 or 100.0. This shows that there are cases where if a node deflates its costs, it might still gain some utility (but not as much under truth-telling), and when it inflates it by too much, it gains nothing because it is too expensive to use in data routing. Note that in the vast majority of cases shown, scaling the elec parameter by 100.0 causes utility to be gained to be zero because

there are other nodes that can be chosen whose costs are not as high. This explains why most nodes have zero utility when elec is scaled by 100.0.

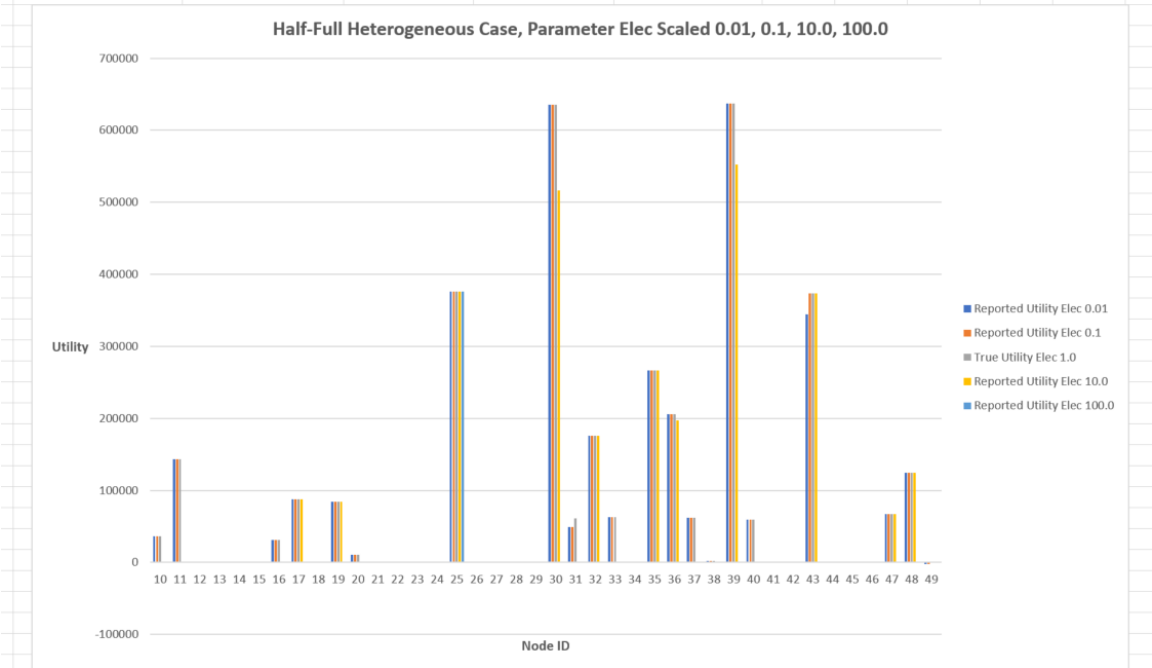


Figure 8. Utilities in the half-full heterogeneous case with scaled elec parameter

Figure 9 shows the utilities in the half-full heterogeneous case when nodes lie about their amp parameter and how the utilities are affected when they are scaled by different factors. For example, Node 36 gains negative utility when it scales its amp parameter by 0.01, which shows that it is being severely punished for suggesting its costs are significantly less than they actually are. When it inflates its costs by 10.0 and 100.0, it gains positive utility, but not as much as under truth-telling. These scenarios show that a node can be punished for inflating as well as deflating its costs and that truth-telling is the best strategy. Note that nodes are affected differently based on their cost parameters. For example, Node 30 gains the same amount of utility when it scales its amp parameter

by 0.01, 0.1, and 1.0 (true value), but experiences decreased utility when it scales it by 10.0 and 100.0. This differs significantly from Node 31, 32, 36, and 47, all of which experience negative utility when they scale their amp parameter by 0.01.

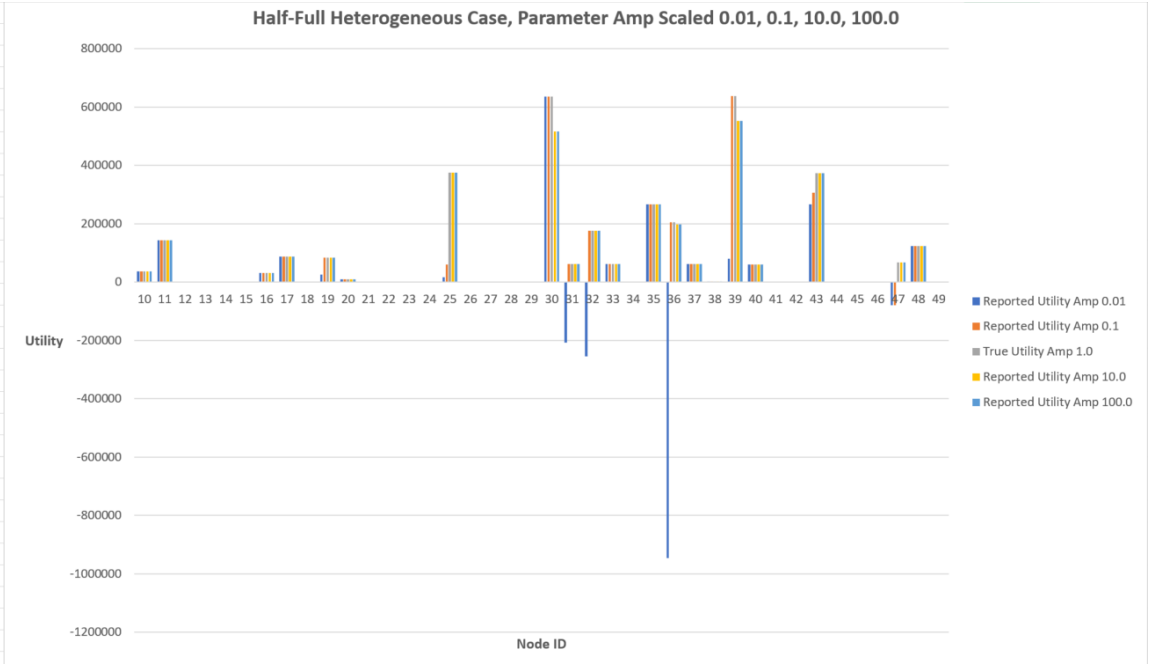


Figure 9. Utilities in the half-full heterogeneous with scaled amp parameter

Figure 10 shows the utilities in the half-full heterogeneous case when nodes lie about their store parameter and how the utilities are affected when they are scaled by different factors. For example, Node 30 gains the same amount of utility when it scales its store parameter by 0.01, 0.1, 1.0 (true cost), and 10.0. However, it gains less than half of the utility it would have gained under truth-telling when it scales the parameter by 100.0.

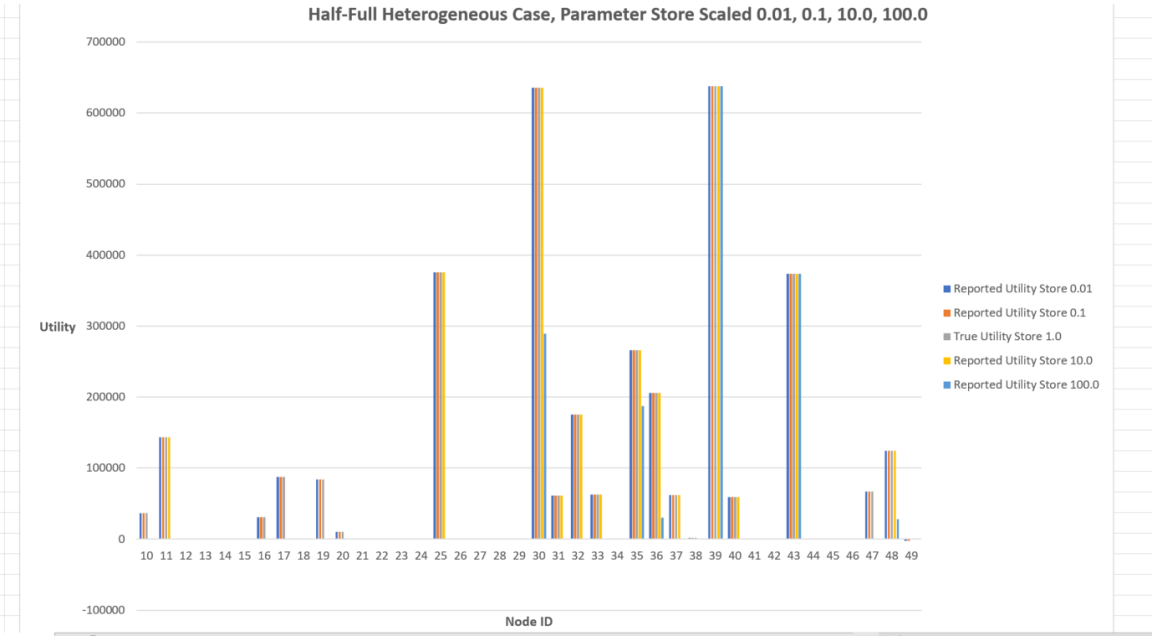


Figure 10. Half-full heterogeneous case with scaled storage parameter

Full Heterogeneous Case

In the full heterogeneous case, each node has randomized cost parameters and all storage nodes except one are filled to capacity. One storage node is unused because the computation of utilities requires that the MCF cost be calculated for a network where one node is removed from the graph.

Figure 11 shows the utilities in the full heterogeneous case when nodes lie about their elec parameter and how the utilities are affected when they are scaled by different factors. For example, when storage node 22 lies about its elec parameter, it acquires slightly less utility when scaling by 0.01, about half of true utility when scaling by 10.0, and zero utility when scaling by 100.0.

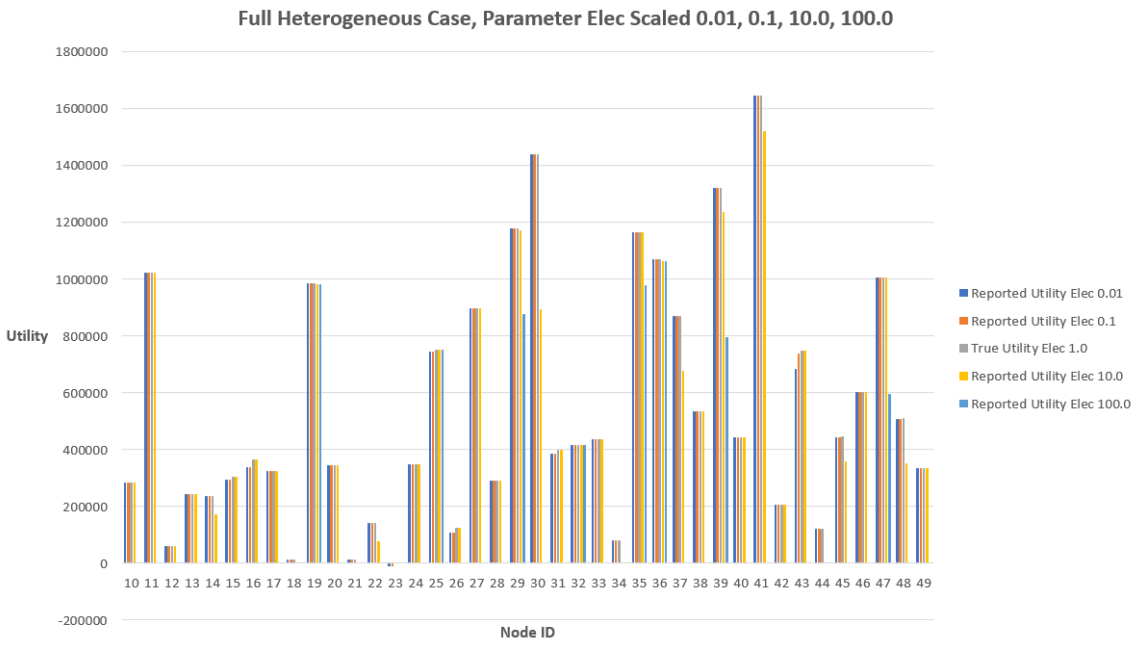


Figure 11. Full heterogeneous case with scaled elec parameter

Figure 12 shows the utilities in the full heterogeneous case when nodes lie about their amp parameter and how the utilities are affected when they are scaled by different factors. For example, Node 14 gains negative utility when scaling its amp parameter by 0.01 and 0.1, and less utility than utility under truth-telling when it scales its parameter by 10.0 and 100.0. This represents a real-world scenario where a node might mistakenly believe that reporting a significantly lower cost would lead to more participation in the network, which might lead to higher utility. However, the node gains negative utility because it still must pay its true costs, but is only compensated for its reported costs, as specified by the payment and utility functions.

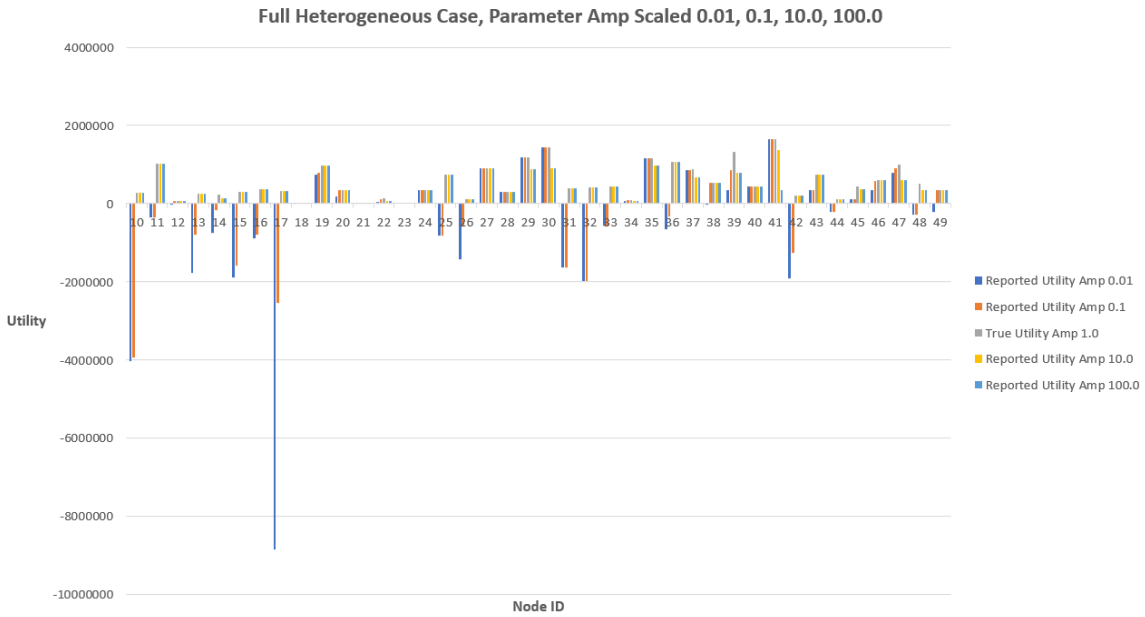


Figure 12. Full heterogeneous case with scaled amp parameter

Figure 13 shows the utilities in the full heterogeneous case when nodes lie about their elec parameter and how the utilities are affected when they are scaled by different factors. For example, when node 22 scales its storage parameter by 10.0 or 100.0, perhaps in a misguided attempt to gain more utility by reporting higher costs, it still acquires less utility than under a truth-telling strategy.

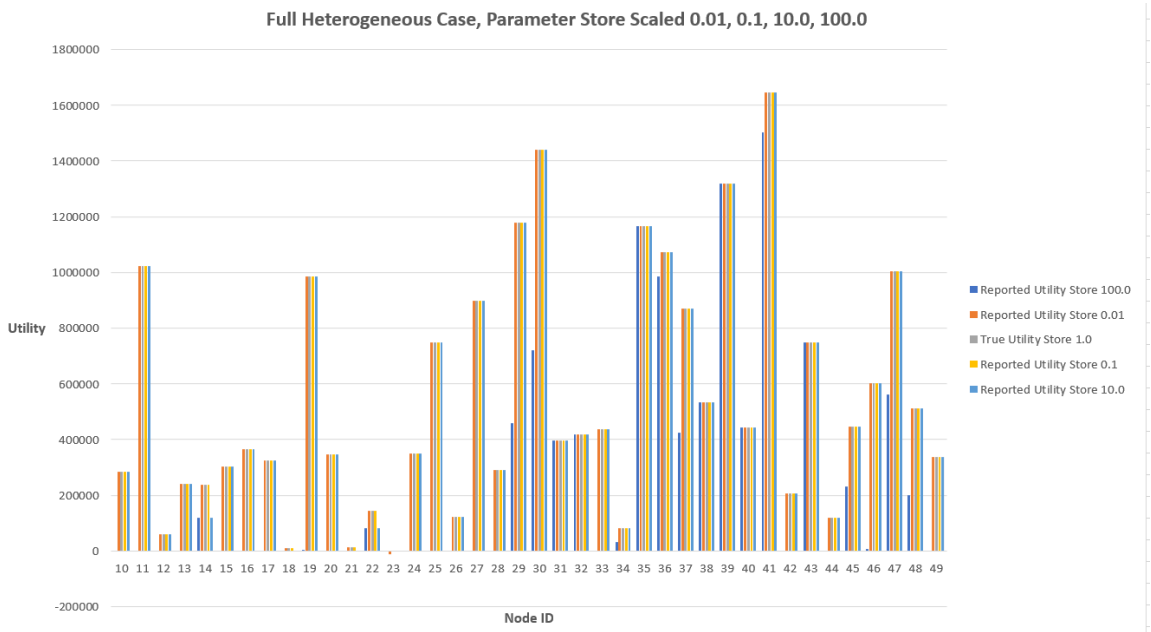


Figure 13. Full heterogeneous case with scaled store parameter

Refinement of Theory

During the development of the simulation for this thesis, an early version of (Chen, Tang, 2016) was consulted as a main reference for the majority of work. Although the simulation produced data that confirmed some aspects of the theory in (Chen, Tang, 2016), it also produced cases that were not analyzed in the paper. In particular, it generated scenarios where the MCF algorithm selects different sets of routing paths based on the nodes' reported costs. The early version of the paper only considered scenarios where the paths did not change. This revealed that there were an additional set of cases that needed to be verified, and additional theoretical results were required to show that these cases still satisfied the claims of the original theory. The most recent version of (Chen, Tang, 2016) has already been edited to reflect these new developments, which were motivated by the results that the simulation produced. This

demonstrates the value of the simulation in this thesis: it can help researchers improve their theory as well as provide evidence for how their theory manifests itself in the real world.

CHAPTER 5

CONCLUSION

This thesis described a simulation that verifies the results in (Chen, Tang, 2016), how to change simulation parameters to model all 21 cases required for successful verification of the results, and how the simulation helped to refine the current theory on non-cooperative data-intensive wireless sensor networks. The next step in the analysis of these networks is to consider more complicated scenarios, such as the infeasible network case (nodes can run out of battery power and disappear from the network), and the collusion scenario (subsets of nodes cooperate only with each other to maximize their utilities). Both of these scenarios are exciting areas of research that will be investigated in the future.

REFERENCES

REFERENCES

Anderegg, L. and Eidenbenz, S. 2003. Ad hoc-VCG: a truthful and cost-efficient routing protocol for mobile ad hoc networks with selfish agents. In Proceedings of the 9th annual international conference on Mobile computing and networking (MobiCom '03). ACM, New York, NY, USA, 245-259.

Candell, R., Liu, Y., Lee, K. Moayeri, N. "A simulation framework for industrial wireless networks and process control systems", Proc. IEEE WFCS'16, pp. 111, May 2016.

Chen, Y. and Tang, B. “Data Preservation in Base Station-less Sensor Networks: A Game Theoretic Approach”, *Proceedings of the 6th EAI International Conference on Game Theory for Networks*, Kelowna, BC, Canada, 2016.

Dobzinski, S. and Nisan, N. 2007. Limitations of VCG-based mechanisms. In Proceedings of the thirty-ninth annual ACM symposium on Theory of computing (STOC '07). ACM, New York, NY, USA, 338-344.

Feigenbaum, J., Papadimitriou C., Sami, R. and Shenker, S. 2005. A BGP-based mechanism for lowest-cost routing. Distributed Computing 18, 1 (July 2005), 61-72.

Feigenbaum, J. and Shenker, S. 2002. Distributed algorithmic mechanism design: recent results and future directions. In Proceedings of the 6th international workshop on Discrete algorithms and methods for mobile computing and communications (DIALM '02). ACM, New York, NY, USA, 1-13.

Felegyhazi, M. and Hubaux, J.-P. "Game theory in wireless networks: A tutorial," École Polytechnique Fédérale de Lausanne. Technical Report LCA-REPORT-2006-002.

Goldberg, A. V., 1997. An efficient implementation of a scaling minimum-cost flow algorithm. J. Algorithms 22, 1 (January 1997), 1-29

Iyengar, S.S and Kannan, R. Game-theoretic models for reliable pathlength and energy-constrained routing with data aggregation in wireless sensor networks, IEEE Journal of Selected Areas in Communications (2004) 1141–1150.

Jaggi, N., Kurkal, R., Tang B., and Wu H. "Energy-Efficient Data Redistribution in Sensor Networks," ACM Transactions on Sensor Networks, v.9, 2013.

Machado, R. and Tekinay, S. 2008. A survey of game-theoretic approaches in wireless sensor networks. Computer Networks Volume 52, Issue 16 (November 2008), 3047-3061.

Nisan, N. and Ronen, A. 1999. Algorithmic mechanism design. In Proceedings of the thirty-first annual ACM symposium on Theory of computing (STOC '99). ACM, New York, NY, USA, 129-140.

APPENDICES

APPENDIX A: Node Summary Files

The following is the data inside a node summary file for both the half-full and full homogeneous cases.

Table 2. Node summary information for half-full and full homogeneous cases.

Node ID	x-coordinate	y-coordinate	elec	amp	store
0	909.2224635	554.4268048	0.001	0.000001	0.001
1	676.9814863	722.2853662	0.001	0.000001	0.001
2	832.7951844	332.178449	0.001	0.000001	0.001
3	507.5877259	396.6751068	0.001	0.000001	0.001
4	636.5041164	84.91108762	0.001	0.000001	0.001
5	441.0072201	571.6131606	0.001	0.000001	0.001
6	435.9609446	443.4736313	0.001	0.000001	0.001
7	994.1529299	649.6453409	0.001	0.000001	0.001
8	710.0784973	953.2690148	0.001	0.000001	0.001
9	234.2105153	195.6704525	0.001	0.000001	0.001
10	590.7941999	624.3671768	0.001	0.000001	0.001
11	766.9457895	213.6255864	0.001	0.000001	0.001
12	167.1260353	678.4533326	0.001	0.000001	0.001
13	291.6671707	559.1702928	0.001	0.000001	0.001
14	225.8332062	846.7684559	0.001	0.000001	0.001
15	549.9003467	594.9673536	0.001	0.000001	0.001
16	512.0856839	492.1435225	0.001	0.000001	0.001
17	484.0895782	567.2282378	0.001	0.000001	0.001
18	160.9302974	903.4245717	0.001	0.000001	0.001
19	818.1482847	210.5121621	0.001	0.000001	0.001
20	423.9732383	342.0313065	0.001	0.000001	0.001
21	73.43917866	772.3372058	0.001	0.000001	0.001
22	195.5822033	808.0867549	0.001	0.000001	0.001
23	241.7721607	989.9827572	0.001	0.000001	0.001
24	129.0438754	37.50102011	0.001	0.000001	0.001
25	951.2658141	548.9329287	0.001	0.000001	0.001
26	325.495177	941.5876492	0.001	0.000001	0.001
27	981.4263854	205.7303347	0.001	0.000001	0.001
28	677.0801524	557.2362663	0.001	0.000001	0.001
29	410.5555609	67.58077626	0.001	0.000001	0.001
30	950.6661551	588.3602947	0.001	0.000001	0.001
31	511.6069573	486.8688972	0.001	0.000001	0.001

32	609.5868356	788.2842582	0.001	0.000001	0.001
33	751.8678442	650.8155353	0.001	0.000001	0.001
34	188.2615175	757.9303341	0.001	0.000001	0.001
35	897.3516312	224.8281603	0.001	0.000001	0.001
36	758.1736675	216.1087046	0.001	0.000001	0.001
37	274.5436804	342.6651528	0.001	0.000001	0.001
38	653.5741547	435.9125845	0.001	0.000001	0.001
39	700.1512086	827.3561562	0.001	0.000001	0.001
40	161.4072592	335.1628165	0.001	0.000001	0.001
41	453.7656901	773.8899734	0.001	0.000001	0.001
42	322.3406721	659.0433282	0.001	0.000001	0.001
43	938.8538011	528.0026772	0.001	0.000001	0.001
44	338.4136694	799.5136259	0.001	0.000001	0.001
45	354.2515728	654.9683081	0.001	0.000001	0.001
46	337.6995201	58.77941388	0.001	0.000001	0.001
47	781.3427602	708.1150704	0.001	0.000001	0.001
48	407.5125966	583.0487686	0.001	0.000001	0.001
49	581.3620954	478.8602166	0.001	0.000001	0.001

The following is the node summary file for half-full and full heterogeneous cases:

Table 3. Node summary file for half-full and full heterogeneous cases.

Node ID	x-coordinate	y-coordinate	elec	amp	store
0	909.2224635	554.4268048	0.099675186	8.86348E-05	0.036194187
1	676.9814863	722.2853662	0.073172859	1.90224E-05	0.041700381
2	832.7951844	332.178449	0.018762011	9.43141E-05	0.091505068
3	507.5877259	396.6751068	0.089660832	6.13652E-06	0.089266933
4	636.5041164	84.91108762	0.097962471	8.41406E-05	0.067284708
5	441.0072201	571.6131606	0.015245709	5.04639E-05	0.098678001
6	435.9609446	443.4736313	0.026550934	9.76092E-05	0.015041567
7	994.1529299	649.6453409	0.032650376	6.67642E-05	0.036964431
8	710.0784973	953.2690148	0.058842165	5.3286E-05	0.021294516
9	234.2105153	195.6704525	0.089971058	3.77146E-05	0.082056177
10	590.7941999	624.3671768	0.047331296	6.92073E-05	0.067300186
11	766.9457895	213.6255864	0.088970032	8.74963E-05	0.059300164
12	167.1260353	678.4533326	0.024406263	6.20793E-05	0.019952437
13	291.6671707	559.1702928	0.082139563	6.15645E-05	0.074813705
14	225.8332062	846.7684559	0.010041498	8.99324E-05	0.070418912
15	549.9003467	594.9673536	0.059040632	3.28444E-05	0.06663868

16	512.0856839	492.1435225	0.086828858	3.90631E-05	0.076934499
17	484.0895782	567.2282378	0.024780998	9.89039E-05	0.064393028
18	160.9302974	903.4245717	0.072391944	6.52007E-05	0.064998225
19	818.1482847	210.5121621	0.02889905	4.49443E-05	0.067578611
20	423.9732383	342.0313065	0.067872004	7.10203E-05	0.038681937
21	73.43917866	772.3372058	0.053793162	6.97888E-05	0.085140488
22	195.5822033	808.0867549	0.037153183	1.57252E-05	0.079807253
23	241.7721607	989.9827572	0.071788992	5.22464E-05	0.060526164
24	129.0438754	37.50102011	0.077030118	7.39184E-05	0.021365486
25	951.2658141	548.9329287	0.010661151	5.41252E-05	0.04493236
26	325.495177	941.5876492	0.048700908	4.95372E-05	0.021859317
27	981.4263854	205.7303347	0.087878993	4.60531E-05	0.07861395
28	677.0801524	557.2362663	0.019036533	9.40945E-05	0.080208231
29	410.5555609	67.58077626	0.025000932	8.86831E-05	0.059966176
30	950.6661551	588.3602947	0.062318654	2.89785E-05	0.044866626
31	511.6069573	486.8688972	0.056171276	5.88874E-05	0.013019621
32	609.5868356	788.2842582	0.010237464	9.19084E-05	0.019253992
33	751.8678442	650.8155353	0.080673818	8.30278E-05	0.026664253
34	188.2615175	757.9303341	0.033922605	1.72329E-06	0.025371008
35	897.3516312	224.8281603	0.037063717	2.73974E-05	0.023217185
36	758.1736675	216.1087046	0.041381791	9.62399E-05	0.043858045
37	274.5436804	342.6651528	0.079560828	1.68593E-05	0.02775392
38	653.5741547	435.9125845	0.053252531	7.31283E-05	0.013711954
39	700.1512086	827.3561562	0.039155467	6.62262E-05	0.00439258
40	161.4072592	335.1628165	0.052117431	2.86327E-05	0.011039327
41	453.7656901	773.8899734	0.025682507	1.7044E-06	0.070533801
42	322.3406721	659.0433282	0.013885942	8.6363E-05	0.019711841
43	938.8538011	528.0026772	0.072242522	2.79464E-05	0.014440761
44	338.4136694	799.5136259	0.069144265	1.74717E-05	0.064629072
45	354.2515728	654.9683081	0.072452555	2.026E-05	0.074988172
46	337.6995201	58.77941388	0.082183172	3.03705E-05	0.094892378
47	781.3427602	708.1150704	0.022087549	2.38212E-05	0.075418425
48	407.5125966	583.0487686	0.057310491	3.59211E-05	0.023747172
49	581.3620954	478.8602166	0.093162155	9.99433E-05	0.049758069

APPENDIX B: Relevant Code Sections

The following are some code sections for the core logic in the simulation. The core logic is the set of steps described in the "Simulation Design" section, partially reproduced below with their corresponding code sections.

Generate Nodes - Generate a set of nodes based either on a user-supplied file or an algorithm that randomly generates a set of nodes.

```
569     /*
570         Generate a collection of randomly generated vertices (nodes) located in a 2D grid.
571         @param numberOfNodes is the number of nodes to generate
572         @return a collection of vertices that represents the randomly generated nodes of a directed graph
573     */
574     public static ArrayList<Vertex> generateNodes(int numberOfNodes) {
575         // make sure there is at least 1 node to generate
576         if (numberOfNodes < 1) {
577             System.out.println("    ERROR: number of nodes must be at least 1");
578             System.exit(1);
579         }
580 
581         // create data structure to hold nodes
582         ArrayList<Vertex> nodes = new ArrayList<Vertex>();
583 
584         // create a node at a random location in 2D space
585         for (int i = 0; i < numberOfNodes; i++) {
586             // generate random x,y coordinates; inclusive on 0.0, exclusive on 1000.0
587             double randomX = ThreadLocalRandom.current().nextDouble(0.0, 1000.0);
588             double randomY = ThreadLocalRandom.current().nextDouble(0.0, 1000.0);
589 
590             // create a 2-dimensional point based on the random coordinates
591             Point2D randomLocation = new Point2D(randomX, randomY);
592 
593             // create a vertex based on the current node ID and the 2D point
594             Vertex randomVertex = new Vertex(i, randomLocation);
595 
596             // add node to collection of nodes that will be returned
597             nodes.add(randomVertex);
598         }
599         return nodes;
600     }
```

Figure 14. Node generation code.

Construct Edges - Based on the transmission range of the nodes provided by the user, construct all pairs of nodes that are within transmission range. The code for this is below:

```
/*
    Generate the edges corresponding to the nodes in the graph
    Note that the cost to route from one node to another will vary depending on storage parameter; compute it separately

    @param nodes is a collection of nodes
    @param dataItemBits is the number of bits per data item
    @return a collection of directed edges that connect the nodes in the graph
*/
public static ArrayList<Edge> generateEdges(List<Vertex> nodes, int dataItemBits) {
    // create an edge if and only if both source and destination can communicate with each other (transmission range allows for it)
    ArrayList<Edge> edges = new ArrayList<Edge>();
    int numberOfNodes = nodes.size();
    for (int i = 0; i < numberOfNodes; i++) {
        // fix a source node
        Vertex currentSource = nodes.get(i);

        // check all other nodes to see if they are within transmission range
        for (int j = 0; j < numberOfNodes; j++) {
            if (i != j) { // exclude self-loop

                Vertex currentDestination = nodes.get(j);

                // get the distance between them and set weight as relay cost
                double currentDistance = currentSource.getDistanceFrom(currentDestination);
                double currentWeight = getRelayCost(currentSource, currentDestination, dataItemBits);

                // verify nodes are within transmission range and create edge between nodes that can transmit to each other
                if ((currentDistance <= currentSource.getTransmissionRange()) && (currentDistance <= currentDestination.getTransmissionRange())) {
                    String currentName = "Node_" + i + "_to_Node_" + j;
                    Edge currentEdge = new Edge(currentName, currentSource, currentDestination, currentWeight);
                    edges.add(currentEdge);
                }
            }
            else {
                // ignore it since it's the same vertex
            }
        }
    }
    return edges;
}
```

Figure 15. Edge construction code.

Construct Graph and Verify Graph Properties - Based on the nodes and edges constructed, construct a graph. Test the graph for biconnectedness. If it is biconnected, proceed. If not, randomly generate nodes and edges until a biconnected graph is constructed. The code for this is below:

```
662     ////////////////////////////////////////////////////////////////////////////
663     //////////////////  CREATE BICONNECETED GRAPH  ///////////////////////////////
664     ////////////////////////////////////////////////////////////////////////////
665
666     /*
667         create a biconnected graph from a node summary file
668
669         @param nodeListPath is the filepath to a node summary file
670         @return a biconnected graph based on the node summary file
671     */
672     public static Graph createBiconnectedGraph(String nodeListPath) {
673
674         System.out.println("    Creating biconnected graph. Standby...");
675         Graph constructedGraph = null;
676
677         // flag to indicate whether simulation needs to be run again
678         boolean rerun_simulation = true;
679
680         // nodes and edges of the graph
681         List<Vertex> nodes = null;
682         List<Edge> edges = null;
683
684         // number of nodes in the graph
685         int numberOfNodes = 0;
686
687         // number of bits for ONE data item
688         int dataItemBits = 512*8;
689
690         while (rerun_simulation == true) {
691
692
```

Figure 16. Construct graph and verify properties – initialization steps.

```
693             ////////////////////////////////////////////////////////////////////
694             ////////////////////////// GENERATE NODES ///////////////////////////
695             ////////////////////////////////////////////////////////////////////
696
697             // generate the nodes based either on a file or randomly generated in this program
698
699             File nodeListFile = new File(nodeListPath);
700             // verify file exists and is not a directory
701             if (nodeListFile.exists() && !nodeListFile.isDirectory()) {
702                 nodes = generateNodesFromFile(nodeListPath);
703                 numberOfNodes = nodes.size();
704                 System.out.println("Number of nodes from file = " + numberOfNodes);
705             }
706             else {
707                 numberOfNodes = 50;
708                 nodes = generateNodes(numberOfNodes);
709             }
710
711             ////////////////////////////////////////////////////////////////////
712             ////////////////////////// GENERATE EDGES ///////////////////////////
713             ////////////////////////////////////////////////////////////////////
714
715             // generate the edges based on the nodes
716             // this is the graph where all nodes tell the truth
717
718
719             edges = generateEdges(nodes, dataItemBits);
720
```

Figure 17. Construct graph and verify properties – node and edge generation.

```
721             ////////////////////////////////////////////////////////////////////
722             ////////////////////////// VERIFY GRAPH PROPERTIES //////////////////
723             ////////////////////////////////////////////////////////////////////
724             constructedGraph = new Graph(nodes, edges);
725             if (constructedGraph.isBiconnected() == false) {
726                 System.out.println(    "Graph is NOT biconnected; re-running simulation");
727                 rerun_simulation = true;
728             }
729             else {
730                 System.out.println(    "Graph IS biconnected.  Continuing simulation...");
731                 rerun_simulation = false;
732             }
733         }
734
735         // graph is now biconnected
736
737         if (constructedGraph == null) {
738             System.out.println("    FATAL ERROR: graph is null!");
739             System.exit(1);
740         }
741
742         // do not allow method to get here if graph is null
743         return constructedGraph;
744     }
```

Figure 18. Construct graph and verify properties – verification of biconnectedness

Compute MCF Input File Under Truth-telling - Based on the biconnected graph, generate an input file that summarizes the graph and node information so that the MCF algorithm

can properly read the information and produce the appropriate output. Assume all nodes tell the truth about their costs. The code for this is below:

```
836     /*
837         generate a text file that will be used as input to the MCF program
838         some metadata information that specifies number of arcs, supply, demand, etc.
839         example line: a <tail> <head> <lower bound capacity> <upper bound capacity> <cost>
840
841         @param nodes is a collection of all the nodes of the graph
842         @param edges is a collection of all the edges of the graph
843         @param dataItemBits is the number of bits per data item
844         @param pathname is the path where the input file will be saved
845     */
846     public static void generateInputMCF(List<Vertex> nodes, List<Edge> edges, int dataItemBits, String pathname) {
847
848         int numberOfNodes = nodes.size();
849
850         // total number of arcs is the total number of edges + 10 extra from virtual source to data generators
851         //     and 40 extra from storage nodes to virtual sink
852         int numberOfEdgesSourceToGenerators = 10;
853         int numberOfEdgesStorageToSink = 40;
854         int numberOfEdgesSourceToStorage = 10 * 40;
855         int numberOfArcs = numberOfEdgesSourceToGenerators + numberOfEdgesSourceToStorage + numberOfEdgesStorageToSink;
856         int totalNodes = numberOfNodes + 2; // includes the virtual source and sink
857         int virtualSourceID = 50;
858         int virtualSinkID = 51;
859
860         // FULL NETWORK CASE
861         int totalSupply = 1950; // 195 data items per data generator, 10 generators total
862         // HALF NETWORK CASE
863         //int totalSupply = 1000; // 1000 total data items
864
865         // start with these lines; "c" denotes "comment"
866         // p min 52 <numberOfArcs>
867         // c min-cost flow problem with 52 nodes and <numberOfArcs> arcs
868         //String str = "MCF_Input_" + timeStamp + ".txt";
869         try
870         {
871             System.out.println("Creating text file with edges and energy costs...");
872
873             // <ID of tail> <ID of head> <capacity lower bound> <capacity upper bound> <cost>
874             FileWriter fstream = new FileWriter(pathname);
875             BufferedWriter out = new BufferedWriter(fstream);
876
877             // p min <number of nodes> <number of edges>
878             out.write("p min " + totalNodes + " " + numberOfArcs + "\n");
879
```

Figure 19. Generate MCF file under truth-telling – initial setup for writing file

```
        // write comment line
        // c min-cost flow problem with 52 nodes and <numberOfArcs> arcs
        out.write("c min-cost flow problem with " + totalNodes + " nodes and " + numberOfArcs + " arcs \n");

        // next lines are
        // n <ID of virtual source> <total supply at source>
        // c supply of <total supply at source> at node <ID of virtual source>
        out.write("n " + virtualSourceID + " " + totalSupply + "\n");
        out.write("c supply of " + totalSupply + " at node " + virtualSourceID + "\n");

        // next lines are
        // n <ID of virtual sink> <total demand of sink>
        // total demand is listed as a negative number
        // so a demand of 1024 is listed as -1024
        // c demand of <total demand at sink, nonnegative> at node <ID of virtual sink>
        int totalDemand = -1*totalSupply;
        out.write("n " + virtualSinkID + " " + totalDemand + "\n");
        // we list demand as nonnegative
        out.write("c demand of " + totalSupply + " at node " + virtualSinkID + "\n");

        // next lines are comments
        //c arc list follows
        //c arc has <tail> <head> <capacity l.b.> <capacity u.b> <cost>
        out.write("c arc list follows \n");
        out.write("c arc has <tail> <head> <capacity l.b.> <capacity u.b> <cost> \n");

        // next set of lines are for virtual source to data generators
        // capacity upper bound is 100 items for each source
        // 512*8*100
        int lowerBoundCapacity = 0;

        ////////////////////////////////////////////////////////////////////////////////
        // FULL NETWORK CASE
        int upperBoundCapacity = 195; // 195 data items from each generator
        ////////////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////////////
        // HALF NETWORK CASE
        //int upperBoundCapacity = 100; // this is for source nodes, 100 data items
        ////////////////////////////////////////////////////////////////////////////////
        int costVirtualSource = 0; // no cost from virtual node to sources
        for (int i = 0; i < 10; i++) {
            out.write("a " + virtualSourceID + " " + i + " " + lowerBoundCapacity + " " + upperBoundCapacity + " " + costVirtualSource + "\n");
        }
```

Figure 20. Generate MCF file under truth-telling – file-writing for source nodes

```
        // execute Dijkstra from source to destination to get the cost
        for (int i = 0; i < 10; i++) {
            // nodes with id 0-9 are the source nodes
            // compute shortest path from each source to each destination

            // get the current data generator, serves as source
            Vertex currentDijkstraSource = nodes.get(i);

            for (int j = 10; j < 50; j++) {
                // start node is a source node
                Vertex currentDijkstraDestination = nodes.get(j);

                // deep copy of edges, but modified
                List<Edge> edges_copy_modified = new ArrayList<Edge>();
                for (Edge edge : edges) {
                    // create deep copy of the edge;
                    Edge currentEdgeCopy = new Edge(edge.getName(), edge.getSource(), edge.getDestination(), edge.getWeight());

                    // modify edge weight if its destination node is final storage node
                    if (edge.getDestination().equals(currentDijkstraDestination)) {
                        // edge destination is the final storage node
                        // set its weight to relay cost + storage cost
                        currentEdgeCopy.setWeight(getFinalStorageCost(edge.getSource(), edge.getDestination(), dataItemBits));
                    }
                    else {
                        // edge weight already correct; do not change
                    }
                    // add the deep copy of the edge to the collection of edges
                    edges_copy_modified.add(currentEdgeCopy);
                }

                // Collection of edges now has the correct weights for this specific arc
                Graph graph = new Graph(nodes, edges_copy_modified);
                DijkstraAlgorithm dijkstra = new DijkstraAlgorithm(graph);
                // this indicates that we are finding all shortest paths
                // starting from data generator node with id "i" (0-9)
                dijkstra.execute(currentDijkstraSource);

                // get shortest path to current storage node
                LinkedList<Vertex> path = dijkstra.getPath(currentDijkstraDestination);

                // compute the total cost of this path
                // extract all edges from the path and get their weights, add together
                double dijkstraCost = 0.0;
```

Figure 21. Generate MCF file under truth-telling – compute source-destination costs

```
968                     for (int k = 0; k < path.size() - 1; k++) {
969                         // path.get(k) is the current source, path.get(k+1) is the current destination
970                         dijkstraCost += getEdgeWeight(path.get(k), path.get(k+1), graph.getEdges());
971                     }
972
973                     // format:
974                     // a <ID of tail> <ID of head> <capacity lower bound> <capacity upper bound> <cost>
975
976                     ////////////////////////////////////////////////////////////////////////////////
977                     // FULL NETWORK CASE
978                     int upperBoundSourceToStorage = 195; // can send max 195 items along path
979                     ////////////////////////////////////////////////////////////////////////////////
980
981                     ////////////////////////////////////////////////////////////////////////////////
982                     // HALF NETWORK CASE
983                     //int upperBoundSourceToStorage = 100; // can send max 100 items along path
984                     ////////////////////////////////////////////////////////////////////////////////
985                     out.write("a " + i + " " + j + " " + lowerBoundCapacity + " " + upperBoundSourceToStorage + " " + dijkstraCost + "\n");
986                 }
987             }
988
989             // final lines summarize arcs from storage nodes to virtual sink
990             // range from 10 to 49 inclusive are storage nodes
991             for (int i = 10; i < 50; i++) {
992                 // a <ID of storage> <ID of virtual sink> <capacity lower bound> <capacity upper bound> <cost>
993                 // cost is 0 because virtual sink is not actually there
994                 int upperBoundCapacityStorage = 50; // 50 items per data node
995                 out.write("a " + i + " " + virtualSinkID + " " + lowerBoundCapacity + " " + upperBoundCapacityStorage + " " + 0 + "\n");
996             }
997
998             out.close();
999             System.out.println("Text file creation complete!");
1000            System.out.println("    Filename: " + pathname);
1001        }
1002        catch (Exception e){//Catch exception if any
1003            System.err.println("Error: " + e.getMessage());
1004        }
1005    }
```

Figure 22. Generate MCF file under truth-telling – connect destination to sink nodes

Compute Preservation Paths - Run the MCF algorithm on the input file to generate a set of paths along which data will be routed. The algorithm also computes the total energy required to route all the data. This is the MCF cost under truth-telling when all nodes participate. The MCF algorithm is a C program that is invoked by the simulation. The code to invoke an executable from the simulation is here:

```
1043    /*  Run an exectuable
1044
1045        @param command is the command to run
1046        @param pathname
1047    */
1048    public static void run_program(String command, String pathname) {
1049        try {
1050
1051            Runtime rt = Runtime.getRuntime();
1052
1053            // bash command line options
1054            // http://tldp.org/LDP/abs/html/bash-options.html
1055            // may need to echo $SHELL first to get the location of the shell to execute it
1056            Process proc = rt.exec(new String[] {"/bin/bash", "-c", command}, null);
1057
1058            // use this to write to a file without using redirect operator
1059            BufferedReader stdInput = new BufferedReader(new InputStreamReader(proc.getInputStream()));
1060            BufferedReader stdError = new BufferedReader(new InputStreamReader(proc.getErrorStream()));
1061
1062            // read the output from the command
1063            String s = null;
1064            // do not omit this; if this is omitted, something incorrect happens when writing files
1065            try
1066            {
1067                FileWriter fstream = new FileWriter(pathname);
1068                BufferedWriter out = new BufferedWriter(fstream);
1069                while ((s = stdInput.readLine()) != null) {
1070                    out.write(s);
1071                }
1072                out.close();
1073            } catch (Exception e) {
1074                e.printStackTrace();
1075            }
1076        }
1077        catch (Exception e) {
1078            e.printStackTrace();
1079        }
1080    }
```

Figure 23. Run an executable from the simulation.

The logic for calling the MCF program from the simulation is here:

```
161        ////////////////////////////////////////////////////////////////////////
162        //////////////////  GENERATE ORIGINAL MCF OUTPUT  //////////////////////
163        ////////////////////////////////////////////////////////////////////////
164
165        String MCF_Output_Original_Path = MCF_Output_Path + "/MCF_Output_ORIGINAL_" + timeStamp + ".txt";
166
167        // cs2 is the executable
168        // "<" operator means "feed input to the executable"
169        // MCF_Original_Path is the path to the original (unmodified) MCF input
170        // ">" operator means "write standard output to particular path"
171        // MCF_Output_Original_Path is the pathname for the output of the MCF program
172        //     summary: run MCF program with input we specify, and output results to path we specify
173        String command = "./cs2-4.6/cs2<" + MCF_Original_Path + ">" + MCF_Output_Original_Path;
174
175        // run the MCF program and generate output file
176        run_program(command, MCF_Output_Original_Path);
```

Figure 24. Invoke MCF program from simulation.

The simulation manually computes the costs based on the MCF program's output to avoid truncation issues present in the C program. This computation is seen here:

```
1791    /*
1792        Compute the MCF cost manually to avoid truncation issues
1793
1794        @param MCF_Output_Filepath is the filepath pointing to the MCF output
1795        @param unmodifiedGraph is the original graph where node parameters have NOT been modified
1796        @paramdataItemBits is the number of bits per data item
1797        @return the MCF cost computed manually, more accurate than original output from MCF program
1798    */
1799    public static double compute_actual_MCF_cost(String MCF_Output_Filepath, Graph unmodifiedGraph, int dataItemBits) {
1800
1801        double actualTotalCost = 0.0;
1802        String currentLine;
1803        try {
1804            BufferedReader br = new BufferedReader(new FileReader(MCF_Output_Filepath));
1805            while ((currentLine = br.readLine()) != null) {
1806                System.out.println("Current line = " + currentLine);
1807                if (currentLine.charAt(0) == 'f') {
1808                    // first character is an f, indicating a flow
1809                    String[] data = currentLine.split("\\s+");
1810                    // data line is now:
1811                    // data[0] = f
1812                    // data[1] = source ID
1813                    // data[2] = target ID
1814                    // data[3] = number of data items
1815
1816                    int sourceID = Integer.parseInt(data[1]);
1817                    int destinationID = Integer.parseInt(data[2]);
1818                    int numDataItems = Integer.parseInt(data[3]);
1819
1820
1821                    // make sure the flow involves the actual nodes rather than virtual source or virtual sink
1822                    if ((Integer.parseInt(data[3]) > 0) && (sourceID != 50) && (destinationID != 51) ) {
1823                        // this is a flow with non-zero data items
1824                        // get the path associated with this source and destination
1825                        List<Vertex> nodes = unmodifiedGraph.getVertices();
1826                        DijkstraAlgorithm dijkstra = new DijkstraAlgorithm(unmodifiedGraph);
1827
1828                        // find all shortest paths from source ID
```

Figure 25. Compute more accurate MCF cost – get source and destination nodes

```
1829                    Vertex sourceVertex = getVertex(nodes, sourceID);
1830                    Vertex destinationVertex = getVertex(nodes, destinationID);
1831                    dijkstra.execute(sourceVertex);
1832                    LinkedList<Vertex> path = dijkstra.getPath(destinationVertex); // get the path to the destination node
1833
1834                    //////////////////////////////////////////////////////
1835                    ///////////  DEBUG: Display Path //////////////////////
1836                    //////////////////////////////////////////////////////
1837                    System.out.println("Least Cost Path from " + sourceID + " to " + destinationID + ":");
1838                    for (Vertex vertex : path) {
1839                        System.out.println(vertex);
1840                    }
1841
1842                    // calculate the total amount of energy required to route along this path
1843                    // we perform a one lookahead because we need the successor node
1844                    // thus, when i is path.size()-2, successor is path.size()-1
1845                    double pathCostPerItem = 0.0;
1846                    for (int i = 0; i < path.size()-1; i++) {
1847
1848                        // use path.get() to get vertex at a particular index in the path
1849                        // current vertex being examined
1850                        Vertex currentVertex = path.get(i);
1851                        Vertex nextVertex = path.get(i+1);
1852
1853                        if (i != path.size() - 2) {
1854                            // we are not at the second-to-last vertex
1855                            // so we can compute relay cost for one data item
1856                            double currentEdgeCostPerItem = getRelayCost(currentVertex, nextVertex, dataItemBits);
1857
1858                            // add the current edge cost to the total path cost
1859                            pathCostPerItem += currentEdgeCostPerItem;
1860                        }
1861                        else {
1862                            // we are at the second-to-last vertex
1863                            double currentEdgeCostPerItem = getFinalStorageCost(currentVertex, nextVertex, dataItemBits);
1864                            pathCostPerItem += currentEdgeCostPerItem;
1865                        }
1866                    }
```

Figure 26. Compute more accurate MCF cost – compute edge costs

```
1867
1868                            // path cost for routing one data item multiplied by total number of data items
1869                            double totalPathCost = pathCostPerItem * numDataItems;
1870                            System.out.println("    Total path cost = " + totalPathCost);
1871                            actualTotalCost += totalPathCost;
1872                        }
1873                        else {
1874                            // ignore
1875                        }
1876                    }
1877                    else {
1878                        // ignore
1879                    }
1880                }
1881            }
1882            catch (Exception e) {
1883                e.printStackTrace();
1884                System.out.println("    Exception in compute_actual_MCF_cost(); exiting...");
1885                System.exit(1);
1886            }
1887
1888            return actualTotalCost;
1889        }
```

Figure 27. Compute more accurate MCF cost – sum all costs

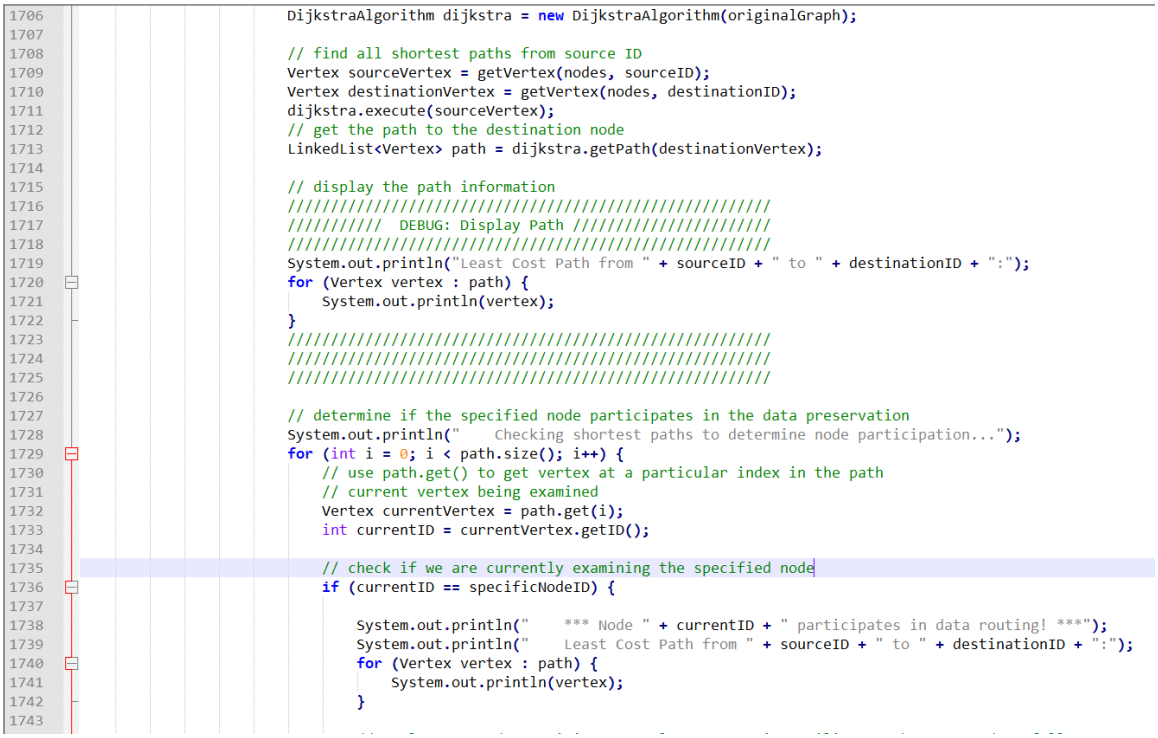
Compute Costs with a Node Removed - The code for this section is the same as the previous MCF cost computation, except that the input graph has a node removed and all

edges incident to this node are removed as well. The underlying logic is the same, so the code is not shown here to avoid unnecessary duplication.

Compute Utilities Under Truth-telling - Compute the utility of each storage node under the truth-telling strategy based on the payment and utility functions in (Chen, Tang, 2016). The utility is given by $\pi_i(\tilde{c}_i, c_{-i}) = c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i$, with symbols defined in Table 1. The code for this computation is here:

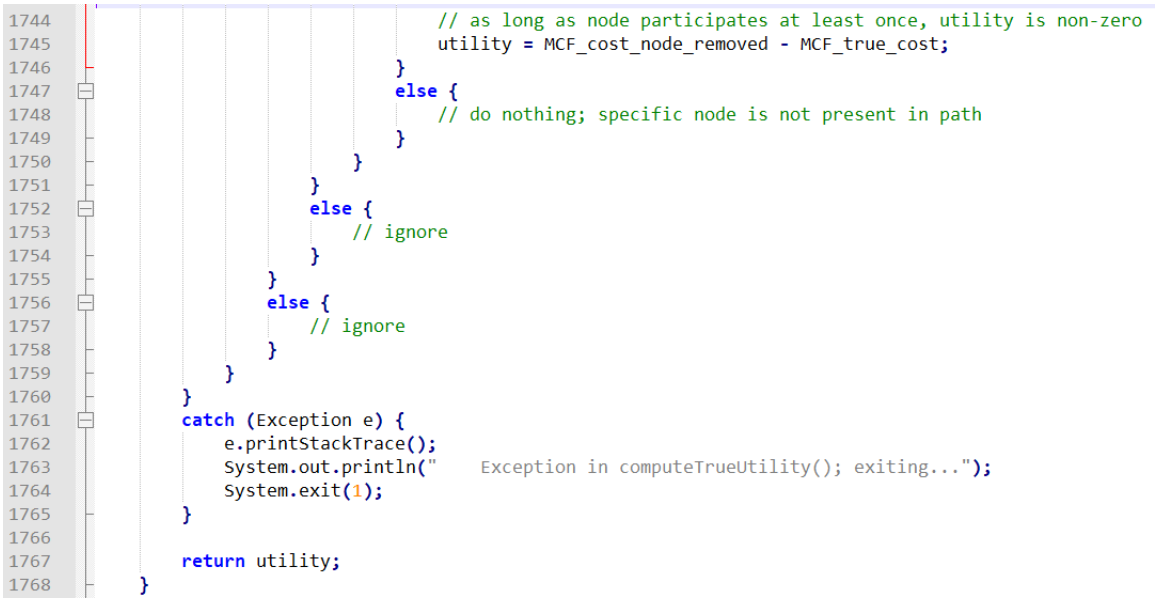
```
1668    /*
1669        Compute utility under truth-telling
1670
1671        @param MCF_cost_node_removed is the MCF cost when the node specified has been removed from the graph
1672        @param MCF_true_cost is the MCF cost when all nodes are present and specified node tells the truth
1673        @param MCF_Output_Original is the path to the original MCF output file (no modifications)
1674        @param originalGraph is the original, unmodified graph
1675        @param specificNodeID is the ID of the node being investigated
1676        @return the utility of a node when it truthfully reports its cost parameters
1677    */
1678    public static double computeTrueUtility(double MCF_cost_node_removed, double MCF_true_cost, String MCF_Output_Original, Graph originalGraph, int specificNodeID) {
1679
1680        System.out.println("Computing TRUE utility of Node " + specificNodeID);
1681        double utility = 0.0;
1682
1683        // read MCF output file line-by-line for costs
1684        String currentLine;
1685        try {
1686            BufferedReader br = new BufferedReader(new FileReader(MCF_Output_Original));
1687            while ((currentLine = br.readLine()) != null) {
1688                if (currentLine.charAt(0) == 'f') {
1689                    // first character is an f, indicating a flow
1690                    String[] data = currentLine.split("\\s+");
1691                    // data line is now:
1692                    // data[0] = f
1693                    // data[1] = source ID
1694                    // data[2] = target ID
1695                    // data[3] = number of data items
1696
1697                    int sourceID = Integer.parseInt(data[1]);
1698                    int destinationID = Integer.parseInt(data[2]);
1699                    int numDataItems = Integer.parseInt(data[3]);
1700
1701                    // make sure the flow involves the actual nodes rather than virtual source or virtual sink
1702                    if ((Integer.parseInt(data[3]) > 0) && (sourceID != 50) && (destinationID != 51) ) {
1703                        // this is a flow with non-zero data items
1704                        // get the path associated with this source and destination
1705                        List<Vertex> nodes = originalGraph.getVertices();
```

Figure 28. Compute utilities under truth-telling – get source and destination nodes



```
1706                    DijkstraAlgorithm dijkstra = new DijkstraAlgorithm(originalGraph);
1707
1708                    // find all shortest paths from source ID
1709                    Vertex sourceVertex = getVertex(nodes, sourceID);
1710                    Vertex destinationVertex = getVertex(nodes, destinationID);
1711                    dijkstra.execute(sourceVertex);
1712                    // get the path to the destination node
1713                    LinkedList<Vertex> path = dijkstra.getPath(destinationVertex);
1714
1715                    // display the path information
1716                    ////////////////////////////////////////////////////
1717                    ///////////  DEBUG: Display Path ////////////////////
1718                    ////////////////////////////////////////////////////
1719                    System.out.println("Least Cost Path from " + sourceID + " to " + destinationID + ":");
1720                    for (Vertex vertex : path) {
1721                        System.out.println(vertex);
1722                    }
1723                    ////////////////////////////////////////////////////
1724                    ////////////////////////////////////////////////////
1725                    ////////////////////////////////////////////////////
1726
1727                    // determine if the specified node participates in the data preservation
1728                    System.out.println("    Checking shortest paths to determine node participation...");
1729                    for (int i = 0; i < path.size(); i++) {
1730                        // use path.get() to get vertex at a particular index in the path
1731                        // current vertex being examined
1732                        Vertex currentVertex = path.get(i);
1733                        int currentID = currentVertex.getID();
1734
1735                        // check if we are currently examining the specified node
1736                        if (currentID == specificNodeID) {
1737
1738                            System.out.println("    *** Node " + currentID + " participates in data routing! ***");
1739                            System.out.println("    Least Cost Path from " + sourceID + " to " + destinationID + ":");
1740                            for (Vertex vertex : path) {
1741                                System.out.println(vertex);
1742                            }
1743
```

Figure 29. Compute utilities under truth-telling – get all shortest paths



```
1744                                // as long as node participates at least once, utility is non-zero
1745                                utility = MCF_cost_node_removed - MCF_true_cost;
1746                            }
1747                            else {
1748                                // do nothing; specific node is not present in path
1749                            }
1750                        }
1751                    }
1752                    else {
1753                        // ignore
1754                    }
1755                }
1756                else {
1757                    // ignore
1758                }
1759            }
1760        }
1761        catch (Exception e) {
1762            e.printStackTrace();
1763            System.out.println("    Exception in computeTrueUtility(); exiting...");
1764            System.exit(1);
1765        }
1766
1767        return utility;
1768    }
```

Figure 30. Compute utilities under truth-telling – utility for participating node

Construct Biconnected Graphs Under a Non-truthful Strategy - The code for this section uses the same logic as the previous biconnected graph construction, except that the node parameters have been altered to reflect a non-truthful strategy. The computation of costs and edge weights still follow the same logic. Thus, the code is not shown here.

Compute Utilities Under a Non-truthful Strategy - Although the logic for this step looks similar to the utility computation under truth-telling, the use of reported costs makes the computation slightly more complicated. Thus, the code section corresponding to this computation is shown here, even though it has many similarities to the computation of utility under truth-telling.

```
1426    /*
1427        Compute utility under a non-truthful strategy
1428        @param MCF_cost_node_removed is the MCF cost when the node is not present in the graph
1429        @param MCF_cost_node_lies is the MCF cost when the node is present, but it lies
1430        @param MCF_Output_Reported is the path to the MCF output for reported costs (node lies)
1431        @param originalGraph is the unmodified graph
1432        @param node_original is the unmodified vertex (no parameters changed)
1433        @param node_modified is the modified vertex (one parameter changed)
1434        @param dataItemBits is the number of bits per data item
1435        @return the utility when a node lies about one of its cost parameters
1436    */
1437    public static double computeReportedUtility(double MCF_cost_node_removed, double MCF_cost_node_lies, String MCF_Output_Reported, Graph originalGraph, Vertex
        node_original, Vertex node_modified, int dataItemBits) {
1438
1439        // verify that the input nodes have the same ID.  If they do not, exit immediately
1440        if (node_original.getID() != node_modified.getID()) {
1441            System.out.println("    FATAL ERROR: When computing reported utility, nodes must have same node ID!  Exiting simulation. ");
1442            System.exit(1);
1443        }
1444
1445        // ID of the node being analyzed
1446        int specificNodeID = node_original.getID();
1447        System.out.println("Computing REPORTED utility of Node " + specificNodeID);
1448
1449        double reportedCost = 0.0;
1450        double trueCost = 0.0;
1451
1452        // read MCF output file for reported costs
1453        // analyze line-by-line
1454        String currentLine;
1455        try {
1456            BufferedReader br = new BufferedReader(new FileReader(MCF_Output_Reported));
1457            while ((currentLine = br.readLine()) != null) {
1458                //System.out.println("Current line = " + currentLine);
1459                if (currentLine.charAt(0) == 'f') {
1460                    // first character is an f, indicating a flow
```

Figure 31. Compute utilities under non-truthful strategy – initialize file reading

```
                String[] data = currentLine.split("\\s+");
                // data line is now:
                // data[0] = f
                // data[1] = source ID
                // data[2] = target ID
                // data[3] = number of data items


                int sourceID = Integer.parseInt(data[1]);
                int destinationID = Integer.parseInt(data[2]);
                int numDataItems = Integer.parseInt(data[3]);

                // make sure the flow involves the actual nodes rather than virtual source or virtual sink
                if ((Integer.parseInt(data[3]) > 0) && (sourceID != 50) && (destinationID != 51) ) {
                    // this is a flow with non-zero data items
                    // get the path associated with this source and destination

                    List<Vertex> nodes = originalGraph.getVertices();
                    DijkstraAlgorithm dijkstra = new DijkstraAlgorithm(originalGraph);

                    // find all shortest paths from source ID
                    Vertex sourceVertex = getVertex(nodes, sourceID);
                    Vertex destinationVertex = getVertex(nodes, destinationID);

                    dijkstra.execute(sourceVertex);
                    LinkedList<Vertex> path = dijkstra.getPath(destinationVertex); // get the path to the destination node

                    // display the path information
                    ////////////////////////////////////////////////////////
                    //////////  DEBUG: Display Path ////////////////////////
                    ////////////////////////////////////////////////////////
                    /*
                    System.out.println("Least Cost Path from " + sourceID + " to " + destinationID + ":");
                    for (Vertex vertex : path) {
                        System.out.println(vertex);
                    }
```

Figure 32. Compute utilities under non-truthful strategy – get shortest paths

```
                    */
                    ////////////////////////////////////////////////////////
                    ////////////////////////////////////////////////////////
                    ////////////////////////////////////////////////////////

                    // determine if the specified node participates in the data preservation
                    System.out.println("    Checking shortest paths to determine node participation...");
                    for (int i = 0; i < path.size(); i++) {

                        // use path.get() to get vertex at a particular index in the path
                        Vertex currentVertex = path.get(i);
                        int currentID = currentVertex.getID();

                        // determine if we are currently looking at node being investigated
                        if (currentID == specificNodeID) {
                            System.out.println("    Node " + currentID + " participates along the path!");
                            System.out.println("    Least Cost Path from " + sourceID + " to " + destinationID + ":");
                            for (Vertex vertex : path) {
                                System.out.println(vertex);
                            }


                            // the specific node we are looking at is participating in data preservation

                            // note that the node should NOT be a transmission node
                            // we assume that transmission nodes are already motivated to participate
                            // CASE 1: node is a relay node
                            // CASE 2: node is a storage node at end of path

                            // vertex is NOT at the end of the path
                            // so we can look one vertex ahead
                            // vertex is a relay node
                            if (i != path.size() - 1) {
                                // CASE 1: node is a relay node
                                System.out.println("    Node " + currentID + " is a relay node.");
                                Vertex nextVertex = path.get(i+1);
```

Figure 33. Compute utilities under non-truthful strategy – check node participation

```
1533                    double distance = currentVertex.getDistanceFrom(nextVertex);
1534
1535                    // reported cost for one data item
1536                    double reportedReceivingCost_perItem = dataItemBits * node_modified.getElec();
1537                    double reportedTransmissionCost_perItem = dataItemBits * node_modified.getAmp() * Math.pow(distance,2) + dataItemBits *
                        node_modified.getElec();
1538                    double reportedCost_perItem = reportedReceivingCost_perItem + reportedTransmissionCost_perItem;
1539                    double reportedCost_thisArc = reportedCost_perItem * numDataItems;
1540                    // accumulate; reported cost is the sum of all reported costs
1541                    reportedCost += reportedCost_thisArc;
1542                    System.out.println("    Reported receiving cost per item = " + reportedReceivingCost_perItem);
1543                    System.out.println("    Reported transmission cost per item = " + reportedTransmissionCost_perItem);
1544                    System.out.println("    Reported relay cost per item = " + reportedCost_perItem);
1545                    System.out.println("    Reported relay cost to route all items (" + numDataItems + ") for THIS SPECIFIC ARC ONLY = " +
                        reportedCost_thisArc);
1546                    System.out.println("    Current accumulated reported relay cost = " + reportedCost);
1547
1548                    // true cost for one data item
1549                    double trueReceivingCost_perItem = dataItemBits * node_original.getElec();
1550                    double trueTransmissionCost_perItem = dataItemBits * node_original.getAmp() * Math.pow(distance,2) + dataItemBits *
                        node_original.getElec();
1551                    double trueCost_perItem = trueReceivingCost_perItem + trueTransmissionCost_perItem;
1552                    double trueCost_thisArc = trueCost_perItem * numDataItems;
1553                    trueCost += trueCost_thisArc;
1554                    System.out.println("    True receiving cost per item = " + trueReceivingCost_perItem);
1555                    System.out.println("    True transmission cost per item = " + trueTransmissionCost_perItem);
1556                    System.out.println("    True relay cost per item = " + trueCost_perItem);
1557                    System.out.println("    True relay cost to route all items (" + numDataItems + ") for THIS SPECIFIC ARC ONLY = " +
                        trueCost_thisArc);
1558                    System.out.println("    Current accumulated true relay cost = " + trueCost);
1559
1560
1561                    // utility for this path
1562                    System.out.println("    MCF cost with node removed = " + MCF_cost_node_removed);
1563                    System.out.println("    MCF cost when node lies = " + MCF_cost_node_lies);
1564                    System.out.println("    Reported cost for this arc = " + reportedCost_thisArc);
```

Figure 34. Compute utilities under non-truthful strategy – compute arc relay costs

```
1565                    System.out.println("    True cost for this arc = " + trueCost_thisArc);
1566                }
1567                else {
1568                    // node is at the end of the path
1569                    // CASE 2: node is a storage node at end of path
1570                    System.out.println("    Node " + currentID + " is a storage node.");
1571
1572                    // compute node's reported costs
1573                    double reportedReceivingCost_perItem = dataItemBits * node_modified.getElec();
1574                    double reportedStorageCost_perItem = dataItemBits * node_modified.getStore();
1575                    double reportedCost_perItem = reportedReceivingCost_perItem + reportedStorageCost_perItem;
1576                    double reportedCost_thisArc = reportedCost_perItem * numDataItems;
1577
1578                    reportedCost += reportedCost_thisArc;
1579
1580                    System.out.println("    Reported receiving cost per item = " + reportedReceivingCost_perItem);
1581                    System.out.println("    Reported storage cost per item = " + reportedStorageCost_perItem);
1582                    System.out.println("    Reported cost to receive and store each item = " + reportedCost_perItem);
1583                    System.out.println("    Reported cost to receive and store all items (" + numDataItems + ") along THIS SPECIFIC ARC ONLY = " +
                         reportedCost_thisArc);
1584                    System.out.println("    Current accumulated reported cost to receive and store = " + reportedCost);
1585
1586                    // compute node's true costs
1587                    double trueReceivingCost_perItem = dataItemBits * node_original.getElec();
1588                    double trueStorageCost_perItem = dataItemBits * node_original.getStore();
1589                    double trueCost_perItem = trueReceivingCost_perItem + trueStorageCost_perItem;
1590                    double trueCost_thisArc = trueCost_perItem * numDataItems;
1591
1592                    trueCost += trueCost_thisArc;
1593
1594                    System.out.println("    True receiving cost per item = " + trueReceivingCost_perItem);
1595                    System.out.println("    True storage cost per item = " + trueStorageCost_perItem);
1596                    System.out.println("    True cost to receive and store each item = " + trueCost_perItem);
1597                    System.out.println("    True cost to receive and store all items (" + numDataItems + ") along THIS SPECIFIC ARC ONLY = " +
                        trueCost_thisArc);
1598                    System.out.println("    Current accumulated true cost to receive and store = " + trueCost);
```

Figure 35. Compute utilities under non-truthful strategy – compute arc final costs

```
                                    // compute utility for routing one data item for this path
                                    System.out.println("    MCF cost with node removed = " + MCF_cost_node_removed);
                                    System.out.println("    MCF cost when node lies = " + MCF_cost_node_lies);
                                    System.out.println("    Reported cost to receive and store all items along this arc = " + reportedCost_thisArc);
                                    System.out.println("    True cost to receive and store all items along this arc = " + trueCost_thisArc);
                                }
                            }
                            else {
                                // do nothing; specific node is not present in path
                            }
                        }
                    }
                    else {
                        // ignore
                    }
                }
                else {
                    // ignore
                }
            }
        }
        catch (Exception e) {
            e.printStackTrace();
            System.out.println("    Exception in computeReportedUtility(); exiting...");
            System.exit(1);
        }

        // when switching between these two equivalent formulas, precision error changes
        //double utility = MCF_cost_node_removed - (MCF_cost_node_lies - reportedCost) - trueCost;
        double utility = MCF_cost_node_removed - MCF_cost_node_lies + reportedCost - trueCost;
        return utility;
    }
```

Figure 36. Compute utilities under non-truthful strategy – compute utility

Collect results and generate summary (CSV file): this is a code section that combines both the collection of results and the generation of a summary file.

```
        System.out.println("Reported utility for Node " + idOfNodeToRemove + " = " + reportedUtility);
        System.out.println("True utility for Node " + idOfNodeToRemove + " = " + trueUtility);
        System.out.println("Comma-separated summary: ");
        System.out.println("Node ID,True Utility,Reported Utility,(True - Reported)");

        double utilityDifference = trueUtility - reportedUtility;
        Results currentResults = new Results(idOfNodeToRemove, trueUtility, reportedUtility);

        System.out.print(currentResults);
        System.out.println("," + utilityDifference);
        results.add(currentResults);
}
        // create CSV file
        System.out.println("Printing all results in CSV format: ");

        // create CSV summary using simulation convention (node index starts at 0)
        String CSV_summary_path = summaryPathname + "/CSV_SUMMARY_" + parameterToModify + "_" + multiplicativeScaleFactor +  ".csv";
        create_CSV_summary(CSV_summary_path, results);

        System.out.println("Node ID,True Utility,Reported Utility,(True - Reported)");
        for (Results currentResult : results) {

            // System.out.println(currentResult);

            double currentTrue = currentResult.getTrueUtility();
            double currentReported = currentResult.getReportedUtility();
            double difference = currentTrue - currentReported;

            System.out.print(currentResult);
            System.out.println("," + difference);
        }
    }
```

Figure 37. Collect results and generate summary CSV file