CHEAT-PROOF DATA PRESERVATION IN BASE STATION-LESS SENSOR NETWORKS

A Project

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

Yu-Ning Yu

Fall 2020

CHEAT-PROOF DATA PRESERVATION IN BASE STATION-LESS SENSOR NETWORKS

AUTHOR: YU-NING YU

APPROVED:

Bin Tang, Ph. D

Project Committee Chair

Liudong Zuo, Ph. D

Committee Member

Mohsen Beheshti, Ph. D

Committee Member

ACKNOWLEDGEMENTS

I would like to thank California State University Dominguez Hills and all the supportive staff and faculty for giving me the chance to complete my master's degree and improving me both academically and personally. I would like to express my great appreciation to Dr. Tang who is my graduation project committee chair, he has great idea and suggestions for my graduation project. Dr. Tang always can give me helpful feedback while I have implementation problem. If I have stuck with understanding algorithm, he always explains to me clearly and patiently. I would also like to thank to Dr. Zuo and Dr. Beheshti, my committee members, who gives me helpful feedbacks to achieve my academic success.

Yu-Ning Yu

Fall 2020

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

We aim to preserve the large amount of data generated inside *base station-less sensor networks* with minimum energy cost, while considering that sensor nodes are selfish. Previous research assumed that all the sensor nodes are cooperative and that sensors have infinite battery power, and designed a centralized minimum-cost flow solution. However, in a distributed setting wherein energy- and storage-constrained sensor nodes are under different control, they could behave selfishly not only to stay away from data preservation but also to lie about their private types in order to maximize their own benefit. In this paper, we show that the traditional VCG mechanism fails to achieve truth-telling in our problem. We design a computationally efficient data preservation game that guarantees cheat-proof outcomes with efficient data preservation. Via extensive simulations, we show that it achieves a system-wide efficient data preservation solution and enforces truth-telling among sensor nodes regarding their private types.

1. INTRODUCTION

Background of Base Station-less Sensor Networks. Sensor networks are ad hoc multi-hop wireless networks formed by a large number of low-cost sensor nodes with limited battery power, storage spaces, and processing capacity. Wireless sensor networks have been used in a wide range of applications such as military surveillance, environmental monitoring, and target tracking [29]. Recently, some of the emerging sensor networks are deployed in challenging environments such as in remote or inhospitable regions, or under extreme weather, to continuously collect large volumes of data for a long period of time. Such emerging sensor networks include seismic sensor networks [8], underwater or ocean sensor networks [23, 14, 28], wind and solar harvesting [15, 5], and volcano eruption monitoring and glacial melting monitoring [27, 16].

In the above scenarios, it is not practical to deploy data-collecting base stations with power outlets in or near such inaccessible sensor fields. Due to the absence of the base stations, these sensor networks are referred to as base station-less sensor networks. Sensory data generated therefore have to be stored inside the network for some unpredictable period of time and then being collected by periodic visits of robots or data mules [21], or by low rate satellite link [9]. In particular, some sensor nodes are close to the events of interest and are constantly generating sensory data, depleting their own storage spaces. We refer to the sensor nodes with depleted storage spaces while still generating data as data nodes. The newly generated data that can no longer be stored at data nodes is called overflow data. To avoid data loss, overflow data is offloaded to sensor nodes with available storages (referred to as storage nodes). We call this process *data preservation in base station-less sensor networks*.

New Challenges in Data Preservation. As wireless communication consumes most of the battery power of sensor nodes, the key challenge of data preservation in base station-less sensor networks is to conserve sensors' battery power. Tang et al. showed that minimizing the total energy consumption of data preservation in base station-less sensor networks is equivalent to minimum cost flow problem [24], which can be solved optimally and efficiently [3]. However, they assume that all the storage nodes are cooperative in the data preservation process. That is, they are all self-less and are willing to contribute their battery power and storage spaces to help offloading and storing the overflow data from the data nodes. In a large-scale distributed sensor networks, however, sensor nodes could be under the control of different users or controllers, each of which pursues their own self-interest. Therefore, sensor nodes can behave selfishly only to maximize their own benefit. Furthermore, sensor nodes are generally resource-constrained, with very limited amount of hardware resources including battery power, storage capacity, and processing power. Such resource constraints give sensor nodes minimum or zero motivation to be an altruistic player in data preservation. In order to conserve their own battery power and storage spaces, the storage nodes will choose not to spend their energy and storage resources to help the data nodes to preserve their overflow data, obstructing the entire data preservation process. When sensor nodes are selfish, the algorithms designed in Tang et al. are no longer valid. The new challenge is how to achieve good system performance, i.e., efficient data preservation with minimum energy cost, while still accommodating selfishness of the sensor nodes.

Algorithmic Mechanism Design (AMDs). In this paper, we address the above challenge by utilizing the technique of **algorithmic mechanism design (AMD)** [17, 19, 18], a subfield of microeconomics and game theory. The goal of AMD is desirable for our data preservation problem – it designs computationally efficient game (including strategies and payoffs) such that individual players (i.e., sensors), motivated solely by self-interest, achieve good system-wide solution. In particular, we consider the Vickrey-Groves-Clark (VCG) mechanism [25, 12, 7]. VCG mechanism is a major branch of AMD that motivates selfish players to participate in games while guaranteeing that each agent truthfully reports its true valuation [17]. In VCG payment model, each storage node needs to be paid in order to be motivated to participate in data preservation. Using VCG, a centralized algorithm calculates the minimized total energy for data preservation based on reported thus possibly misleading cost types. For all the nodes chosen to participate in data preservation, the centralized algorithm designates each with either data relaying or data storing tasks. Thus, for each node, the VCG payment model guarantees that it is optimal truthfully reporting its private type, compared to lying in order to drop out of data preservation or to switch to different data preservation tasks.

However, several complexities in our model deems the applicability of VCG mechanism directly into our problem not feasible. For example, VCG mechanism looks at optimization problems where each agent either participates in certain project or not as the outcome. Instead, in our work when a storage node receives a data packet originated from a data node, it faces three different outcomes: it could relay a data packet, or store a data packet, or not participate in data preservation. Therefore, each node could have subtle incentives to lie about its private type in order to switch the outcome for itself from one to the other, so long as doing so improves its utility. In light of the exist. We will address a few challenges applying VCG directly into our problem and propose our solutions.

What Is Working? First, we show that the existing VCG model works for the data preservation process wherein sensor nodes have infinite amount of energy power. In this scenario, VCG achieves below two guarantees that for all the participating sensor nodes. First, each node, understanding how the payments are calculated, finds that truthfully reporting its private cost information is an optimal strategy. Second, based on the reported cost of each node, the VCG payment can sufficiently motivate each node to actually participate in data preservation. In particular, VCG guarantees that truth-telling is a dominate strategy; that is, the truth-telling utility of a node is always greater than or equal to its lying utility, making a node to always opt for truth-telling strategy. With these two goals achieved, the VCG payment model in our game leads to optimal system-wide data preservation solution in terms of energy cost with each sensor node motivated solely by self-interest.

What Is Not Working? However, we observe that when nodes have finite amount of energy, VCG payment model does not deliver truth-telling as dominate strategy for some nodes. As the associated costs for data preservation of each storage node (i.e, energy costs of receiving, saving, and transmitting packets) are normally private information which are not directly observed by outsiders, the storage nodes, being selfish, can lie about their costs in order to gain more payment thus more utilities following the VCG payment model. We show that through lying about its associated cost of data preservation, the storage node may successfully induce a data preservation path which generates itself a higher payoff compared to the payoff when it tells the truth. Such lying behavior of the storage nodes out of their selfishness clearly not only makes the data preservation in the network suboptimal and inefficient, and more importantly, makes the VCG model not longer valid in this scenario.

We take a thorough investigation about the cause of above phenomenon with the following finding. As a node lies and exaggerates about its efficacy of processing data packets in the data preservation process, the system, according to the centralized data preservation algorithms, tends to assign the lying node more data packets than what it is actually able to process. Such node has to discard those extra amount of data packets that are beyond its processing capability, resulting in data loss in the network. We come up with a simple fix to the flawed VCG model and show in our experiment results that not only truth-telling becomes the dominate strategy for sensor nodes, but also it achieves optimal data preservation in the network while accommodating the selfish behavior of sensor nodes.

Paper Organization. The rest of the paper is organized as follows. Section 2 review all the related work. Section 3 formulate the data preservation problem. Section 4 present the algorithms for both infinite and finite energy cases.

We then consider the case when each node is energy constrained but the whole network is still feasible for data preservation. In this case, the data preservation route which minimizes total system-wide energy will result in some data loss because of the limited energy of some nodes. While we modify the centralized algorithm to achieve minimum energy subject to energy constraint, we show that a modified payment model is also needed, and it continues to guarantee each node's truth-telling and participation in data preservation.

2. RELATED WORK

Game theory techniques have been extensively applied to solve research problems in computer networks in general and wireless ad hoc and sensor networks in particular [20, 22, 4, 6]. There are three main classes of game theory techniques that are generally employed. Non-cooperative game theory studies strategies between interactions among individual competing players, with Nash Equilibrium (NE) being its solution concept that describes a steady state condition for the players. Cooperative game theory models situations in which players form groups (i.e., coalitions) rather than acting individually. One of its central notations is *the core*, which is the payoff allocation that no group of players has an incentive to leave its coalition to form another coalition. The third one is called cooperation enforcement games [4], wherein selfish players are incentivized to cooperate in order to maximize the social optimal of the system.

3. PROBLEM FORMULATION OF DATA PRESERVATION PROBLEM

Network Model. Network Model. The sensor network is represented as an undirected connected graph $G(V, E)$, where $V = \{1,2,\ldots,n\}$ is the set of n sensor nodes and E is the set of m edges. The sensory data are modeled as a sequence of data packets, each of which is a bits. Some sensor nodes are close to the event of interest and generate large amount of data packets and deplete their storage spaces; they are referred to as *data nodes*. WLOG there are k data nodes $V_S = \{1, 2,\ldots, k\}$. The rest nodes in $V - V_S = \{k + 1, k + 2,\ldots, n\}$ are referred to as *storage nodes*. Let d_i denote the number of overflow data packets data node i generates. Because of the storage depletion of the data nodes, the overflow data packets must be offloaded from their data nodes to some storage nodes to be preserved. Let $d = \sum_{i=1}^{k} d_i$ be the total number of overflow data packets, and let $D = \{D_1, D_2,\ldots, D_d\}$ denote the set of these d data packets. Let $s(j) \in V_S, 1 \leq j \leq d$, denote D_j's data node. Let m_i be the available free storage space (in bits) at sensor node $i \in V. If\ i \in V_S$, then $m_i = 0$, implying that a data node is storage-depleted and thus has zero available storage space. If $i \in V - V_S$, then $m_i \geq 0$, implying that a storage node i can store another m_i bits of data packets. We assume that $\sum_{i=k+1}^{n} m_i \geq d \cdot a$, that is, the total size of the overflow data packets can be accommodated by the total available storage spaces. Fig. 1(a) shows a linear sensor network $G\ (V, E)$ with two data nodes 1 and 3, each having two data packets to offload, and two storage nodes 2 and 4, each having two storage spaces.

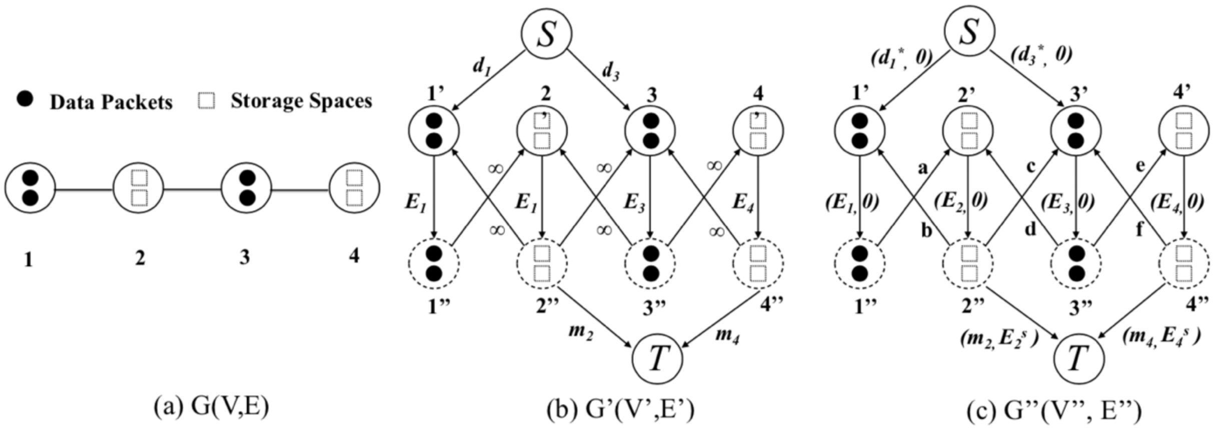


Figure 1: (a) shows a linear sensor network $G\ (V, E)$ with two data nodes 1 and 3, each having two data packets to offload, and two storage nodes 2 and 4, each having two storage spaces. (b) shows its transformed flow network $G\ '(V', E')$ that finds maximum number of data packets to offload. (c) shows its transformed flow network $G\ ''(V'', E'')$that finds the minimum energy cost, given that maximum d1∗ and d3∗ amount of data packets can be offloaded from data node 1 and 3, respectively. Here, $a = (\infty, E_1^t(2) + E_2^r),\ b = (\infty, E_2^t(1) + E_1^r),\ c = (\infty, E_2^t(3) + E_3^r),\ d = (\infty, E_3^t(2) + E_2^r),\ e = (\infty, E_3^t(4) + E_4^r),\ f = (\infty, E_4^t(3) + E_3^r)$.

Energy Model. We consider three different kinds of energy consumptions incurred in data preservation.

- *Transmitting Energy* $E_i^t(j)$. When node is ends a data packet of a bits to its one-hop neighbor j over their distance $l_{i,j}$, the amount of *transmitting energy* spent by i is $E_i^t(j) = a \cdot \epsilon_i^a \cdot l_{i,j}^2 + a \cdot \epsilon_i^e$. Here, ϵ_i^a is energy consumption of sending one bit on transmit amplifier of node $i \cdot \epsilon_i^a = 100pJ/bit/m^2$; and ϵ_i^e is energy consumption of transmitting one bit on the circuit of node $i \cdot \epsilon_i^e = 100nJ/bit$. Note that E_t depends on not only the distance between nodes but also on the size of the data it transmits.
- *Receiving Energy* E_i^r. When node i receives an a-bit data packet from one of its one-hop neighbor, the amount of *receiving energy* it spends is $E_i^r = a \cdot \epsilon_i^e$. Here, ϵ_i^e is energy

consumption of receiving one bit on the circuit of node i. Note that E_i^r only depends on the size of the data it receives, not the distance between nodes.

- *Storing Energy* E_i^s. When node i stores a-bit data into its local storage, the amount of *storing energy* it consumes is $E_i^s = a \cdot \epsilon_i^s$. Here ϵ_i^s is the energy consumption of storing one bit at node i; $\epsilon_i^s = 100nJ/bit$. Thus E_s depends on size of data it stores.

In the baseline model, we assume that each node has enough energy to participate the data preservation process. We study the case when some nodes are subject to energy depletion in data preservation in Section 6.

Our energy model generalizes the well-known first order wireless radio model [13] in two aspects. First, first order model does not consider storing energy parameterized by ϵ_i^s. Mathur et al. [2] examined the energy consumptions of different currently available flash memory, a viable storage technology for low-power, energy-constrained wireless sensor networks. They found that read, write, and erase energy consumption per byte for Hitachi MultiMedia Cards (MMC) and NAND flash memory are $1.108\ \mu J$ and $0.062\ \mu J$ respectively (which are equivalent to $139\ nJ/bit$ and $8\ nJ/bit$, respectively). Therefore, the energy consumption of storing data packets on sensor nodes cannot be neglected. Second, first order radio model assumes that ϵ_i^a and ϵ_i^e are the same for any sensor node i (in particular, $\epsilon_i^a = 100pJ/bit/m^2$ and $\epsilon_i^e = 100nJ/bit$). However, Wang and Yang [26] pointed out that the energy consumption is a function of the features of devices. Specifically, they found that the energy consumption on circuits varies significantly from state to state of a device and among different types of real sensor devices. Thus, in this paper, we assume that different sensor nodes could have different energy parameters. That is, all the three parameters ϵ_i^a, ϵ_i^e, and ϵ_i^s are node dependent.

Problem Formulation. Define a *preservation function* as $p: D \rightarrow V - V_s$, indicating that a data packet $D_j \in D$ is offloaded from its data node $s(j) \in V_s$ to a storage node $p(j) \in V - V_s$ to be preserved. Let $P_j = \{s(j), \dots, p(j)\}$ be the *preservation path* along which D_j is offloaded. Let $c_{i,j}$ denote node $i's$ energy consumption in preserving $D_j \cdot c_{i,j}$ can be represented as Equation 1 below, with $\sigma(i,j)$ being the successor node of i on P_j.

$$c_{i,j} = \begin{cases} E_i^t(\sigma(i,j)) & i = s(j) \\ E_i^r + E_i^s & i = p(j) \\ E_i^r + E_i^t(\sigma(i,j)) & i \in P_j - \{s(j), p(j)\} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The objective is to find a preservation function p and $Pj\ (1 \leq j \leq d)$ to minimize the *total preservation cost*, denoted as c, i.e.,

$$c = min_p \sum_{j=1}^{d} \sum_{i=1}^{n} c_{i,j} = min_p \sum_{i=1}^{n} \sum_{j=1}^{d} c_{i,j}, \tag{2}$$

under the storage constraint that the total size of data offloaded to storage node i cannot exceed $i's$ storage capacity: $|j|1 \leq j \leq d, p(j) = i| \cdot a \leq m_i, \forall i \in V - V_s$.

4. ALGORITHMIC SOLUTIONS OF DATA PRESERVATION

In this section, we first consider that each node has infinite amount of energy and propose a minimum cost flow-based optimal solution. We then consider that each node has finite energy power and propose a ILP-based optimal solution.

4.1. Data Preservation With Infinite Energy

Minimum Cost Flow Solution. Let $w(i,j) = E_i^t(j) + E_j^r$ be the total energy consumption when node i sends an a-bit packet to its one-hop neighbor j; i.e., $w(i,j)$ is $i's$ transmitting energy plus $j's$ receiving energy. Given a data node v_1 and a storage node v_x, let $P(v_1, v_x) = \{i, v_1, v_2, \dots, v_x\}, (v_i, v_{i+1}) \in E, 1 \leq i \leq x-1$, denote a *shortest path* between v_1 and v_x (this can be computed efficiently using Dijkstra's algorithm). Let $c = (v_1, v_x)$ denote the total energy consumption spent on sending one data packet from v_1 to v_x along $P = (v_1, v_{x)}$. That is, $c = (v_1, v_x) = \sum_{i=1}^{x-1} w(v_i, v_{i+1}) + E_{v_x}^s$. It includes the transmitting energy of $(v_1$, the receiving energy as well as transmitting energy of $v_i,\ 2 \leq i \leq x-1$, and the receiving energy as well as storing energy of v_x.

Tang. et al. [24] has shown that this problem is equivalent to the minimum cost flow problem in a flow network $G\ (V', E')$ properly transformed from the sensor network graph $G\ (V, E)$. The transformation takes place as shown in Fig. 2.

Step I. $V' = \{s\} \cup \{t\} \cup V_S \cup V - V_S$, where s is the data node and t is the sink node in the flow network.

Step II. $E' = \{(s,i)\} \cup \{(i,j)\} \cup \{(j,t)\}$, where $i \in V_S$ and $j \in V - V_S$. Note that it is a complete bipartite graph between V_S and $V - V_S$.

Step III. For each edge (s,i), set its capacity as d_i and cost as 0. For each edge (j,t), set its capacity as m_j, the storage capacity of j, and cost as 0.

Step IV. For each edge (i,j), set its capacity as d_i and cost as $c_{i,j}$. Here v is the total minimum energy consumption sending one data packet from data node i to storage node j.

Step V. Set the supply at s and the demand at t as $\sum_{i=1}^{k} d_i$.

However, as $c(v_1, v_x)$ indicates the energy consumption on the shortest path between v_1 and v_x, above minimum cost flow formulation implicitly assumes that each node on such shortest path has enough energy to participate the data preservation process. The minimum cost flow problem can be solved optimally and efficiently [3]. We adopt and implement the scaling push-relabel algorithm proposed in [10, 1]. It has the time complexity of $O(|V|^2|E|log(|V|C))$, where C is the maximum capacity of an edge in the transformed graph. We denote the algorithm designed in Tang. et al. [24] as *the centralized algorithm* to highlight that it minimizes data preservation energy based on the assumption that each node in the network is selfless and therefore fully cooperative.

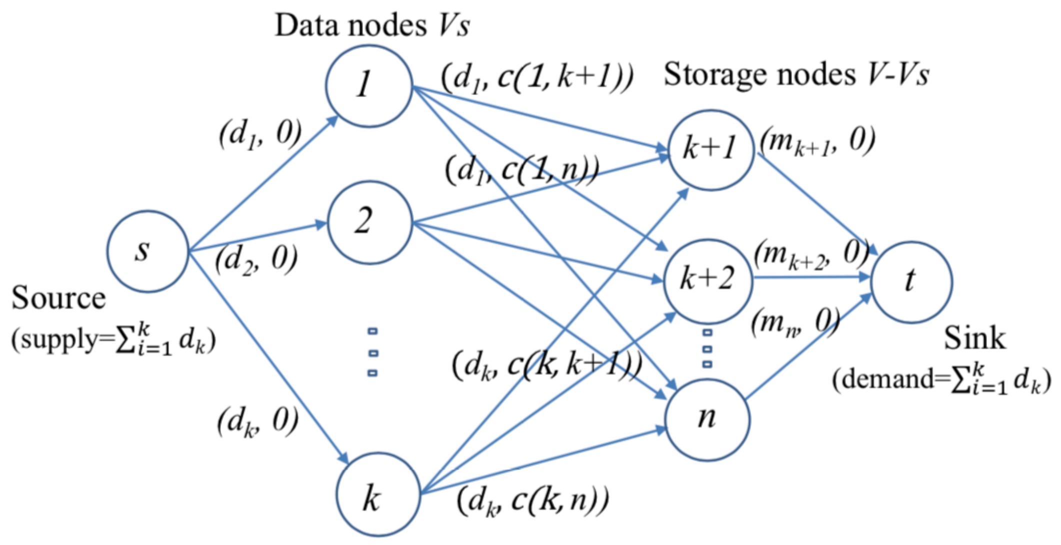


Figure 2: Minimum cost flow model.

4.2. Data Preservation With Finite Energy

In this section, we will first present another related problem called feasibility problem. We will then formulate an ILP to solve data preservation optimally.

4.2.1. Data Preservation Feasibility Problem.

When sensor nodes have limited amount of initial battery energy, it is possible that some of them can exhaust and deplete their energy and data packets can no longer be transmitted along the shortest path between data nodes and storage nodes. Thus, a new question arises: Given a sensor network topology with data nodes and storage nodes, and each node has an initial finite energy level, is it possible that all the data packets from the data nodes can be offloaded? We refer to this problem as *data preservation feasibility problem*. Before we answer this question, we first transform the sensor network $G\ (V, E)$, shown in Fig. 1(a), to a flow network $G\ '(V', E')$, shown in Fig. 1(b).

1) Replace each undirected edge $(i, j) \in E$ with two directed edges (i, j) and (j, i). Set the capacities of all the directed edges as infinity.
2) Split node $i \in V$ into two nodes: *in-node* i' and *out-node* i''. Add a directed edge (i', i'') with capacity of E_i, the initial energy level of node i. All the incoming directed edges of node i are incident on i' and all the outgoing directed edges of node i emanate from i''. Therefore, the two directed edges (i, j) and (j, i) in Step 1) are now changed to (i'', j') and (j'', i').
3) Add a super data node *S*, and connect *S* to the in-node i' of the data node $i \in V_S$ with an edge. Set the capacity of this edge as d_i, the number of data packets at data node i.
4) Add a super sink node *T*, and connect out-node j'' of the storage node $j \in V - V_S$ to *T*. Set its edge capacity m_j, the storage capacity of storage node *j*.

Therefore, $V' = \{S\} \cup \{T\} \cup \{i' : i \in V\} \cup \{i'' : i \in V\}$ and $E' = \{(i'', j') : (i, j) \in E\} \cup \{(j'', i') : (i, j) \in E\} \cup \{(i', i'') : i \in V\} \cup \{(S, i') : i \in V_s\} \cup \{(j'', T) : j \in V - V_s\}$. We have $|V'| = 2n + 2\ and\ |E'| = 2m + 2n.$

Next, we formulate and solve a linear program on the flow network $G\,'(V', E')$ in Fig. 1(b), in order to find the maximum amount of data packets that can be offloaded in the sensor network in Fig. 1(b). Let x_{ij} be the amount of flows on edge (i, j) in $G\,'(V', E')$.

$$maximize \sum_{i \in V_s} x_{S_i'}$$

$$\text{subject to} \sum_{j:e_i \in S_j} x_j \geq 1\,, \qquad i = 1, \dots, n$$

$$x_j \in \{0,1\}, \qquad j = 1, \dots, m$$

$$x_{S_{i'}} \leq d_i, \qquad i \in V_s$$

$$x_{i''T} \leq m_i, \qquad i \in V - V_s$$

$$x_{S_{i'}} += m_i, \qquad i \in V_s$$

$$maximize \sum_{i \in V_s} x_{S_i'} \tag{3}$$

$$s.t\;\; x_{S_{i'}} \leq d_i, \qquad i \in V_s \tag{4}$$

$$x_{i''T} \leq m_i, \qquad i \in V - V_s \tag{5}$$

$$x_{S_{i'}} + \sum_{j:(i,j) \in E} x_{j''i'} = \sum_{j:(i,j) \in E} x_{i''j'}\,, \qquad i \in V_s \tag{6}$$

$$\sum_{j:(i,j) \in E} x_{j''i'} = \sum_{j:(i,j) \in E} x_{i''j'} + x_{i''T}, \qquad i \in V - V_s \tag{7}$$

$$E_i^r \times \sum_{j:(i,j)\in E} x_{j''i'} + \sum_{j:(i,j)\in E} (E_i^t(j) \times x_{i''j'}) \leq E_i\,, \qquad i \in V_s \tag{8}$$

$$E_i^r \times \sum_{j:(i,j)\in E} x_{j''i'} + \sum_{j:(i,j)\in E} E_i^t(j) \times x_{i''j'} + E_i^s \times x_{i''T} \leq E_i, \qquad i \in V - V_s \tag{9}$$

Here, $\sum_{i\in V_s} x_{S_i'}$ in Objective (3) is to find maximum amount of packets that can be offloaded in the entire network. Inequality (4) indicates the number of packets data node i can offload is less than or equals d_i, the initial number of data packets data node i has. Inequality (5) indicates the maximum number of packets storage node i can store is m_i, the storage capacity of storage node i. Equation (6) shows the flow conservation for data nodes, where the number of its own data packets offloaded plus the number of data packets it relays for other data nodes equals the number of data packets it transmits. Equation (7) is the flow conservation for storage nodes, which says that data packets a storage node receives are either relayed to other nodes or stored by this storage node. Inequalities (8) and (9) represents the energy constraints for data nodes and storage nodes respectively and need some special note. In particular, each data node costs its energy power when it transmits its own data packets as well as relays (i.e., receives and transmits) data packets for other data nodes; each storage node costs its energy when it stores or relays data packets from data nodes.

Note that above formulation is different from that of the classic maximum flow problem. While classic maximum flow stipulates that the amount of flows on each edge cannot exceed its edge capacity, in our formulation, the amount of flow on edge (i', i'') (i.e., $x_{i'i''}$) and the capacity on edge(i', i'') (i.e., E_i) do not follow this relationship. Instead, the relationship between $x_{i'i''}$ and E_i is more intricate. We observe that for data node i, $x_{i'i''}$ equals to the amount of data packets i

can offload (i.e., $x_{S_{i'}}$) plus the amount of data packets it relays for other data nodes (i.e., $\sum_{j:(i,j)\in E} x_{j''i'}$). $x_{i'i''}$ also equals to the total amount of data packets it transmits (i.e., $\sum_{j:(i,j)\in E} x_{i''j'}$). The energy cost of sensor node i can now be expressed as a function of $x_{S_{i'}}$ and $x_{j''i'}$ (r.h.s. of Inequality (8)). Similarly, for storage node i, $x_{i'i''}$ equals to the amount of data packets i can store (i.e., $x_{i''T}$) plus the amount of data packets it relays for other data nodes (i.e., $x_{j''i'}$), while the energy cost of i is a function of $x_{i''T}$ and $x_{j''i'}$ (r.h.s of Inequality (9)). Finally, a sensor node (be it a data node or storage node) $i's$ energy cost must be less than or equal to its initial energy level E_i, thus giving the correlation between $x_{i'i''}$ and E_i.

4.2.2. Minimum Cost Flow ILP Solution

After finding the number of packets that can be offloaded by each data node, say, data node i can offload d_i^* data packets out of its original d_i data packets, next it needs to compute the minimum energy consumption spent for offloading these data packets. We accomplish this by formulating and solving another linear program. To do that, we first transform the sensor network $G(V, E)$, shown in Fig. 1(a), to a flow network $G''(V'', E'')$, shown in Fig. 1(c). The graph topology of $G''(V'', E'')$ looks similar to that of $G'(V', E')$ in Fig. 1(b). However, as this is a minimum cost flow formulation, each edge in Fig. 1(c) has a capacity as well as a cost, which are specified as below (from top to bottom).

1) For directed edge connecting super data node S to the in-node i' of the data node $i \in V_S$, set its capacity as d_i^*, the number of data packets that are computed by the maximum flow formulation, set its cost as zero.

2) For directed edge (i', i''); set its capacity as E_i, the initial energy level of node i, and cost as zero.
3) For directed edge (i'', j'), set its capacity as infinity and cost as $E_i^t(j) + E_j^r$, the sum of node $i's$ transmitting energy and node j's receiving energy. For directed edge (j'', i'), set its capacity as infinity and cost as $E_j^t(i) + E_i^r$, the sum of node $j's$ transmitting energy and node $i's$ receiving energy.
4) For directed edge connecting the out-node i'' of the storage node $j \in V - V_s$ to super sink node T, set its capacity as m_i, the storage capacity of i, and its cost as E_i.

Now, we find the minimum energy cost of offloading d_i^* packets from data node i, $\forall i \in V_s$ by formulating and solving a linear program upon $G''(V'', E'')$, as shown below. Here x_{ij} is the amount of flow on edge $(i,j) \in E''$ and c_{ij} is the cost of one amount of flow on edge (i,j), thus $\sum_{(i,j)\in E''} x_{ij} \times c_{ij}$ is the total energy consumption in the entire network. In particular, for edge (i'', j'), its cost $c_{i'',j'} = E_i^t(j) + E_j^r$, which equals the sum sensor node $i's$ transmitting energy and sensor node$j's$ receiving energy; for edge (j'', i'), its cost $c_{j'',i'} = E_j^t(i) + E_i^r$, which equals the sum of sensor node $j's$ transmitting energy and sensor node $i's$ receiving energy. Note that its constraints are very similar to those for the maximum flow formulation, except Equation (11), as the number of offloaded data packets is given in minimum cost flow linear programming formulation.

$$maximize \sum_{(i,j)\in E''} x_{ij} \times C_{ij} \tag{10}$$

$$s.t\ x_{S_{i'}} \leq d_i^*, \qquad i \in V_s \tag{11}$$

$$x_{i''T} \leq m_i, \qquad i \in V - V_s \qquad (12)$$

$$x_{S_{i'}} + \sum_{j:(i,j)\in E} x_{j''i'} = \sum_{j:(i,j)\in E} x_{i''j'}\,, \qquad i \in V_s \qquad (13)$$

$$\sum_{j:(i,j)\in E} x_{j''i'} = \sum_{j:(i,j)\in E} x_{i''j'} + x_{i''T}, \qquad i \in V - V_s \qquad (14)$$

$$E_i^r \times \sum_{j:(i,j)\in E} x_{j''i'} + \sum_{j:(i,j)\in E} (E_i^t(j) \times x_{i''j'}) \leq E_i\,, \qquad i \in V_s \qquad (15)$$

$$E_i^r \times \sum_{j:(i,j)\in E} x_{j''i'} + \sum_{j:(i,j)\in E} E_i^t(j) \times x_{i''j'} + E_i^s \times x_{i''T} \leq E_i, \qquad i \in V - V_s \qquad (16)$$

Again, note that above linear programming formulation is different from that of the classic minimum cost flow problem, for the same reason discussed in maximum flow case.

In this work, we instead consider selfishness of nodes in the sense that each node is maximizing its own interest instead of the system interest. The central problem is to design a mechanism to incentivize selfish nodes to accomplish data preservation as in the centralized algorithm. Note that each data node is obligated to offload its data therefore selfishness does not apply to data nodes. On the other hand, storage nodes are selfish and need to be motivated. However, selfishness of storage nodes can lead to two problems. First, each storage node has no incentive to either relay or store data as either task consumes energy. Therefore, our mechanism needs to pay those storage nodes involved in data preservation path solved from the centralized algorithm, in order to give them incentive to participate in data preservation. The second problem is more subtle but fundamental. The centralized algorithm can figure out the minimum cost data preservation path only based on the assumption that data preservation costs of each storage node are observed. However, some of those cost parameters of each node (given by ϵ_i^e, $\epsilon_i^a$and ϵ_i^s) are private information of each node and may not be directly observed by outsiders. Thus, our mechanism needs to induce each node to truthfully report their unobserved cost parameters, so that the centralized algorithm can calculate the minimum cost path based on the reported cost parameters.

5. ALGORITHMIC MECHANISM DESIGN (AMD) APPROACH

The goal of AMD is to design a game in which selfish players maximizing their own utility will choose strategies resulting in the social optimum specified by an optimal algorithm. Here the resulted state is referred to as the *dominant strategy equilibrium/solution*. Dominant strategy of a player is a strategy always maximizing his utility regardless of the other players' strategies. In a dominant strategy solution, each player is playing his dominant strategy. Note that a dominant strategy solution is also a Nash equilibrium since no player has an incentive to deviate from its strategy unilaterally. The challenge in the data preservation problem is to design utility function so that truthfully reporting its cost parameter is a *dominant strategy* to each storage node. Below we first introduce the concepts and notations of the AMD model. We then present the payment model, and prove that under this payment model, acting truthfully (that is, telling its true energy cost involved in data preservation) is each node's dominant strategy.

The AMD Model. There are n nodes in the network - node i has some private information t_i, called its type. There is an *output specification* that maps each type vector $t = \{t_1, \dots, t_n\}$ to some output o. Node i's cost is given by *valuation function* $v_i(t_i, o)$, which depends on t_i as well as o. A *mechanism* defines for each node i is a set of strategies A_i. When i plays strategy $a_i \in A_i$, the mechanism computes an *output* $o = o(a_1, \ldots, a_n)$ and a *payment vector* $p = (p_1, \ldots, p_n)$, where $p_i = p_i(a_1, \ldots, a_n)$. Node i wants to maximize its utility function $\pi_i(a_1, .., a_n) = v_i(t_i, c) + p_i$.

Recall that C_{ij} is energy cost of sensor node i in preserving data packet D_j, which is given by Equation 1. $C_{ij}$is private information to node i because all of the cost parameters viz. $\epsilon_i^e, \epsilon_i^a$and $\epsilon_i^s$are unobservable to the public. Therefore, we define $t_i \in \{\epsilon_i^e, \epsilon_i^a, \epsilon_i^s\}$ as node i's private type. Node $i's$ strategy set A_i includes any value of private type t_i it can report; and $v_i(t_i, o) = -C_{ij}$ given by (1). Its utility is $\pi_i(t_i, t_{-i}) = p_i - \sum_{j=1}^{d} c_{i,j}$, where d is the total number of data packets to be offloaded.

According to the total preservation cost given by equation (2), to preserve a single data packet, the total cost in the network is the sum of all participating nodes' energy costs. We can there- fore consider the Vickrey-Groves-Clark (VCG) mechanism [25, 12, 7]. VCG mechanism is a major branch of mechanism design which applies to optimization problems where the objective function is simply the sum of all agents' valuations, and it guarantees that each agent truthfully reports its true valuation [17]. However, several complexities in our model deems the applicability of VCG mechanism not clear: first, VCG mechanism looks at optimization problems where each agent either participates in certain project or not as the outcome. Instead, in our work each node faces three different outcomes for each data packet according to the centralized algorithm: it could relay a data packet, or store a data packet, or not participate in data preservation. Therefore, each node could have subtle incentives to lie about its private type in order to switch the outcome for itself from one to the other, so long as doing so improves its utility. In light of the existence of triple out- comes to each agent for each data packet, a closer examination is required on the applicability of VCG mechanism. Second, when there are multiple overflow data packets, efficient data preservation cost in the network is not a simple summation of the minimized preservation cost of each data packet D_j due to storage capacity constraint of each storage node. In order to present a detailed analysis, we shall first study our payment model based on a single overflow data packet; then we extend to multiple overflow data packets and show that our results continue to hold. Third, the baseline model assumes no energy constraint, i.e., each node is with enough energy to carry out its tasks in data preservation assigned by the centralized algorithm. However, if nodes are subject to energy constraint such that some may eventually die while carrying out data preservation, the original VCG mechanism could fail to achieve social efficiency. We show that this indeed is the case. In addition, we propose a modified VCG mechanism, which again guarantees truthfulness to each node and therefore restores data preservation efficiency to the network.

Payment and Utility Model. Below we present the payment and utility model. We use c_i to denote node $i's$ true total cost in data preservation, and p_i the total payment to node i. Thus $\pi_i = p_i - c_i$. Let $t_{-i} = \{t_1, \ldots.., t_{i-1}, t_{i+1}, \ldots, t_n\}$ denote the vector of cost types of all other nodes except node i, and $c_{-i} = \{c_1, \ldots.., c_{i-1}, c_{i+1}, \ldots, c_n\}$ denote the data preservation costs of all other nodes except node i.

Definition 1. (**Payment and Utility.**) Based on Green and Laffont [11], under VCG mechanism, given any cost $\tilde{t}_i$reported by node i, the amount of payment given to node i depends on whether node i is chosen to participate in data preservation according to the centralized algorithm. Its payment is 0 if it is not chosen; and its payment when it is chosen is:

$$p_i(\tilde{t}_i, t_{-i}) = C_{v-\{i\}} - (\tilde{c}_V - \tilde{c}_i), \tag{17}$$

where $c_{V-\{i\}}$ is the minimum total cost of the preservation path that does not go through i; $\tilde{c}_V$ is the minimum total cost of the preservation path that goes through i, when i reports its cost $\tilde{c}_i$.

Therefore, $i's$ utility is 0 when it is not chosen by the centralized algorithm; and when i is chosen, its utility is

$$\pi_i(\tilde{t}_i, t_{-i}) = p_i(\tilde{t}_i, t_{-i}) - c_i = c_{v-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i, \tag{18}$$

where c_i is node $i's$ true cost. Moreover, we define c_v as the minimum total cost of the preservation path that goes through i when i truthfully reports its cost, i.e., when $\tilde{t}_i = t_i$.

Time complexity of the payment model. The time taken to compute the payment model is the time taken for the minimum cost flow calculation, which is $O(|V|^2|E|log(|V|C))$, where C is the maximum capacity of an edge in the transformed graph [10, 1]. Under this model, the amount of payment given to a specific node i equals the total minimum cost of all the participating nodes when i does not participate minus all other participating nodes' cost when i participates. The rationale is that a node can be motivated to participate if it is paid its share of contribution, which in our case, is the amount of preservation energy this node helps to reduce when it participates.

An implication here is that the payment and utility model is common knowledge to each node. That is, each node understands that based on their reported cost types and the corresponding data preservation path calculated by the centralized algorithm, their payment and utility are given by (17) and (18), respectively. The timing of the game among the data nodes is given below.

Definition 2. (**Timing of the Game.**) The game unfolds as follows. In stage 1, each storage node reports its private type t_i. In stage 2, the centralized algorithm is applied based on reported cost types to calculate the minimum cost data preservation path. In stage 3, each of the storage nodes chosen in the path chooses to participate in data preservation or not. If they participate, they realize the data preservation cost and also the payment given by Equation (17), and each gets utility given by Equation (18).

Note that each storage node moves only in stages 1 and 3, when each chooses how much to report for private type and whether to participate in data preservation based on the corresponding payment. Stage 2 is non-strategic: in the absence of base stations, the centralized algorithm is provided by an outsider of the system, and it cannot be enforced in the system by the outsider. Since there is a time sequence between the two decisions of each node in stage 1 and stage 3, the solution concept of the game is subgame perfect Nash equilibrium (SPNE). SPNE is a Nash equilibrium (NE) in which players are doing NE in every subgame of the whole game tree.

The payment model aims at achieving the following two properties:

1. Individual-rationality (**IR**). It is the participation constraint which makes sure that each node, when truthfully reporting its cost type, will participate in data preservation once it is chosen by the centralized algorithm. That is,

$$\pi_i(t_i, t_{-i}) \geq 0 \ \forall t_{-i} \text{ and } \forall_i \in V - V_s.$$

2. Incentive-compatibility (**IC**). It requires that truth fully reporting private cost type is the dominant strategy of each node. Namely, each node gets the highest utility under truth-telling regardless of reported types of other nodes:

$$\pi_i(t_i, t_{-i}) \geq \pi_i(\tilde{t}_i, t_{-i}) \ \forall t_{-i}, \forall \tilde{t}_i \neq t_i \text{ and } \forall_i \in V - V_s.$$

Note that when IR and IC are satisfied, it is a dominant strategy solution to our game that each node truthfully reports its private type, then participates in data preservation whenever chosen by the centralized algorithm.

Assumptions. We make several assumptions in the baseline model. First, the data nodes are obliged to offload their overflow data packets to other storage nodes and need not to be motivated. Their types are public knowledge, and they will be reimbursed according to their true costs entailed in data preservation. Second, the payment model is common knowledge to each node. That means each node understands that based on reported cost types, the centralized algorithm will calculate the efficient data preservation path; correspondingly, its payment and utility are given by (17) and (18) if it participates in data preservation, or zero if it does not. Third, no single storage node is *critical* to the data preservation. This means that for each node, if it is removed from the network, data preservation shall still be feasible to the network. Feasibility means that the network has the capability (in terms of storage capacity and energy) to preserve all the over-flow data packets. The above assumptions are needed for VCG mechanism to work. Forth, each node, although is subject to storage capacity constraint, is not energy-constrained when performing data preservation tasks. This last assumption will be relaxed in Section 6, where we consider the payment model when each node is subject to energy constraint.

For storage node i that participates in the preservation of a specific data packet, it incurs one of the two costs below:

- *Relaying Cost* $c_i^r(j)$. When node i receives a data packet and then sends it to one of its one-hop neighbor j over their distance $l_{i,j}$, its *relaying cost*, denoted as $c_i^r(j)$, is the sum of its receiving energy and transmitting energy. That is $c_i^r(j) = E_i^r + E_i^t(j) = 2 \cdot a \cdot \epsilon_i^e + a \cdot \epsilon_i^a \cdot l_{i,j}^2$.
- *Storing Cost* c_i^s. When node i receives a data packet and then stores it into its storage, its *storing cost*, denoted as c_i^s, is the sum of its receiving energy and its storing energy. That is, $c_i^s = a \cdot \epsilon_i^e + a \cdot \epsilon_i^s$.

Note that for each node i, either ϵ_i^a or ϵ_i^s or ϵ_i^e is its private type, among which ϵ_i^e is involved in the calculation for both $c_i^r(j)$ and c_i^s, while ϵ_i^a is involved only in the calculation for $c_i^r(j)$ and ϵ_i^s is involved only for c_i^s. There are three different designated roles to each node: either it does not participate in data preservation; or when it participates, it may either relay or store the data packet. By lying about its type, node i might switch its role from one to the other among these three roles. In what follows, we consider each cost parameter as nodes' private type to examine whether truthfulness satisfies IR and IC in the payment model. Note that given a reported type vector $t = \{t_1, \ldots, t_n\}$, it could occur that there exist multiple routes which minimize the data preservation cost. If that is the case, in reality the centralized algorithm randomly picks up one route. W.L.O.G., at a given $t = \{t_1, \ldots, t_n\}$, we treat all the (feasible) data preservation routes which generate the same total preservation cost to the network thereby making the centralized algorithm indifferent as "the same data preservation routes". Thus, the term "different data preservation routes" refers to the routes with strictly different data preservation cost for the network.

An assumption of our work is that the network is feasible in terms of storage capacity for data preservation, i.e., $\sum_{i=k+1}^{n} m_i \geq d \cdot a$ (the available storage spaces in the network is enough for the whole overflow data packets). However, each storage node is subject to its storage capacity constraint $a \leq m_i \leq d \cdot a$. Therefore, minimizing the total preservation cost of multiple data packets cannot be reduced to a simple aggregation of minimizing the preservation cost of each single data packet.

The following graph illustrates a simple network wherein minimum total data preservation cost is not a linear summation of minimum preservation cost of each data packet. In the network, $S1$ and $S2$ are two data nodes and nodes 1 and 2 are two storage nodes. All cost parameters of nodes 1 and 2 are the same; each has a capacity to store one unit of overflow data packet. The distance between the two data nodes to node 1 are 1; the distance between *S*2 to node 2 is 2. Consider the scenario when there is a single overflow data packet, generated either by *S*1 or *S*2. In either case, the minimum cost preservation route is to send the overflow data packet to node 1 for storage. Instead, suppose *S*1 and *S*2 each generates a overflow data packet so that there are two overflow data packets in the network. Due to the storage capacity constraint of node 1, the minimum cost data preservation requires *S*1 to send its data to node 1 for storage and *S*2 to send its data to node 2 for storage. The total minimized preservation cost is clearly not a summation of the minimized preservation cost with single overflow data packet in the network.

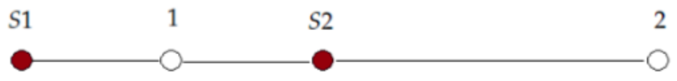


Figure 3: A Network with Multiple data nodes

As shown by the example, the centralized algorithm is a synchronized procedure, which calculate optimal preservation paths of multiple data packages while taking into account storage capacity of each storage node. On the other hand, the total energy cost for data preservation of the network is a linear summation of the energy cost of each single storage node in data preservation. In what follows, we show that when nodes are not energy constrained, the VCG mechanism will motivate each node to truthfully report its private type. Before we prove this result, we first present the following lemma.

Lemma 1. *At given t_{-i}, if the centralized algorithm designates the same tasks to node i under $\hat{t}_i$ and $\tilde{t}_i$ with $\hat{t}_i \neq \tilde{t}_i$, it must be that the preservation routes of all data packets remain the same under $\hat{t}_i$ and $\tilde{t}_i$.*

Proof: We need to prove that all the nodes other than i are doing the same tasks under $\hat{t}_i$ and $\tilde{t}_i$, when node i are designated the same tasks. Denote the minimized total data preservation cost $c_v = c_i + c_{-i}$, with c_{-i} the data preservation cost of all nodes except node i. Denote $\hat{c}_v = \hat{c}_i + \hat{c}_{-i}$ under $(\hat{t}_i , t_{-i})$, $\tilde{c}_v = \tilde{c}_i + \tilde{c}_{-i}$ under $(\tilde{t}_i , t_{-i})$. It holds that $\hat{c}_i = \tilde{c}_i$. Suppose data preservation routes are different hence $\hat{c}_{-i} \neq \tilde{c}_{-i}$. Suppose $\hat{c}_{-i} < \tilde{c}_{-i}$ W.L.O.G.. Given that node i conducts same tasks, the data preservation routes under $(\hat{t}_i , t_{-i})$ must also be available under $(\hat{t}_i , t_{-i})$. Thus under $(\tilde{t}_i , t_{-i})$, switching to the same data preservation routes found under $(\hat{t}_i , t_{-i})$ gives a total cost $\tilde{c}_i + \hat{c}_{-i} < \tilde{c}_i + \tilde{c}_{-i} = \tilde{c}_v$, a contradiction to cost minimization of the centralized algorithm.

Theorem 1. *Under the payment given by (17) and* $t_i \in \{\epsilon_i^a, \epsilon_i^s, \epsilon_i^e\}$, *both IR and IC are satisfied. It is a dominant strategy of every storage node* i *to truthfully report its cost type in stage 1; then to participate in data preservation in stage 3 if node* i *is chosen by the centralized algorithm.*

Proof: Node i can either report truthfully or tell a lie about its cost type $t_i \in \{\epsilon_i^a, \epsilon_i^s, \epsilon_i^e\}$. According to a comparison between the outcome for i under truthfulness and under lying, there can be four cases. Below we show that IR and IC are satisfied in all the four cases.

Case I: Node i is not in the preservation path when reporting either t_i or $\tilde{t}_i$. In this case $\pi_i(t_i, t_{-i}) = \pi_i(\tilde{t}_i, t_{-i}) = 0$

Case II: Node i is in the preservation path when reporting t_i, which implies that $c_{V-\{i\}} \geq c_V$; and it is not in the preservation path when reporting $\tilde{t}_i$, which gives payoff $\pi_i(\tilde{t}_i, t_{-i}) = 0$. Thus, its payoff under truth-telling is $\pi_i(t_i, t_{-i}) = c_{V-\{i\}} - c_V \geq 0$. In this case $\pi_i(t_i, t_{-i}) \geq \pi_i(\tilde{t}_i, t_{-i})$.

Case III: Node i is not in the preservation path when reporting t_i; however, it is in the preservation path when reporting $\tilde{t}_i$. Thus $\pi_i(t_i, t_{-i}) = 0$. Denote $\tilde{c}_V^{t_i}$ as the true data preservation cost (based on (t_i, t_{-i})) when data preservation route is determined under $(\tilde{t}_i, t_{-i})$. We have $c_{V-\{i\}} \leq \tilde{c}_V^{t_i}$ due to cost minimization under t_i. Node $i's$ payoff under $\tilde{t}_i$ is $\pi_i(\tilde{t}_i, t_{-i}) = c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i = c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i + c_i) = c_{V-\{i\}} - \tilde{c}_V^{t_i} \leq 0$. Thus $\pi_i(t_i, t_{-i}) \geq \pi_i(\tilde{t}_i, t_{-i})$.

Case IV: Node i is in the preservation path when reporting either t_i or $\tilde{t}_i$. There are different subcases depending on the tasks assigned to node i. We discuss each subcase below:

Subcase IVa. Node i is assigned exactly the same tasks when reporting either t_i or $\tilde{t}_i$. By Lemma 1, node i is getting the same payoff since $\pi_i(\tilde{t}_i, t_{-i}) = c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i = c_{V-\{i\}} - c_V = \pi_i(t_i, t_{-i}) \geq 0$.

Subcase IVb. The jobs assigned by the centralized algorithm to node i are different under t_i and $\tilde{t}_i$. The payment of i when it reports truthfully is $\pi_i(t_i, t_{-i}) = c_{V-\{i\}} - (c_V - c_i) - c_i = c_{V-\{i\}} - c_V \geq 0$. Instead when it lies by reporting $\tilde{t}_i$, its payoff is $\pi_i(\tilde{t}_i, t_{-i}) = c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i = c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i + c_i)$. Here $\tilde{c}_V - \tilde{c}_i + c_i \equiv \tilde{c}_V^{t_i}$, the total preservation cost calculated according to (t_i, t_{-i}) using the route found by the centralized algorithm under $(\tilde{t}_i, t_{-i})$. By cost minimization of the centralized algorithm under $(t_i, t_{-i}), \tilde{c}_V^{t_i} \geq c_V$. Thus $\pi_i(\tilde{t}_i, t_{-i}) \leq c_{V-\{i\}} - c_V = \pi_i(t_i, t_{-i})$.

Since IR and IC are satisfied, the results immediately follow.

To see the intuition, note that the idea of the VCG mechanism is to give each node a net payoff (utility) according to its marginal contribution in data preservation. Whenever lying by node i leads to different data preservation routes, it is always possible switching the preservation routes under truthfulness to the ones found under node $i's$ lying. The reason is that nodes are not energy-constrained; therefore, switching between different routes will not result in any data loss. Since lying by a node will increase the real total data preservation cost in the network, it will reduce the marginal contribution made by the node to the network, implying a lower utility to the node. As a result, the VCG mechanism will provide incentives for each node to tell the truth and to participate in data preservation.

However, the above argument will no longer hold true when nodes are energy constrained. We consider this scenario in the following section.

6. STORAGE NODES WITH ENERGY CONSTRAINT

Note that our baseline model assumes that each node is subject to no energy constraint. In this section, we consider the scenario when storage nodes are energy-constrained and may not have enough energy to conduct data-preservation tasks assigned by the centralized algorithm. On the other hand, the whole network is still *feasible* in terms of energy for data preservation. That is, total system energy of the network is enough for the preservation of overflow data packets. i.e., $\sum_{i=k+1}^{n} e_i \geq \sum_{i=k+1}^{n} c_i$.

The centralized algorithm shall be correspondingly modified to take into consideration both the capacity and the energy constraints. Here we focus on the payment model. When nodes are also energy constrained, multiple problems could arise and make the original payment model no longer work. We explain each of the problem below.

First, we need to deal with data loss in multiple cases. In one case, consider that a node exaggerates its private cost by reporting a higher cost type. The centralized algorithm may find the network infeasible to store all overflow data packets under reported cost types. In another case, note that the VCG payment calculation requires the value of c_{-i}, the total minimized data preservation cost when node i does not exist. However, the network system could become infeasible for data preservation when a node is removed from the network. In either case, the centralized algorithm needs to deal with the infeasibility of the network for data preservation. In addition, we need to modify the payment model to reflect the system cost due to data loss.

When the network is infeasible, the centralized algorithm will conduct two steps of tasks: step one, solve the problem of maximizing the number of nodes which can be stored by the network; step two, find the cost-minimizing data preservation routes for the data packets chosen to be stored in step one. All the other data packets deemed beyond the system energy capacity will be dropped. By doing so, the centralized algorithm guarantees the overall optimality of the network on data preservation.

Second, a more serious problem is that now a node could have incentives to either lie about its private type, implying that the VCG mechanism may fail to satisfy IC or/and IR. The failure of the VCG mechanism is due to the potential data loss when notes are under energy constraints. If there is no data loss involved in the application of the payment model when nodes are subject to energy constraints, the argument in Theorem 1 continues to hold, and truth-telling continues to be the dominant strategy of each node. Therefore, it is the occurrence of data loss in the procedure of the payment calculation that could result in the violation of IC in the payment model. Below we argue that there are scenarios when a node has incentive to either exaggerate its private type, or downsize its private time.

To see that a node may want to downsize its private type in certain scenarios, we consider a simple example. Consider the network in Fig. 4, where node 0 is the single source node with 5 units of overflow data packets. Node 1 and node 2 are the storage nodes with equal distance given by 1 to node 0, and each are with storage capacity for at least 5 data packets.

Let $a = 1$ for each data packet. Suppose $\epsilon_i^a = 0, \epsilon_i^e = 0$ for $i = 1, 2$, whereas $\epsilon_i^s = 1, \epsilon_i^s = 2$. I.e., nodes 1 and node 2 has zero relaying cost; storing cost of each data packet is 1 for node 1 and 2 for node 2. Thus ϵ_i^s is the private type of each node. In addition, each node is subject to an energy constraint: node 1 has a total energy of 10 and node 2 has a total energy of 4. This means that node 2 can store up to 2 units data packets and node 1 can store up to 10 units data packets.

Under truth-telling, node 2 will not be chosen to store any data packet and its utility is $\pi_2(t_1, t_2) = 0$. Instead, consider node 2 lies by reporting $\tilde{t}_2 = 0.8$. Since node 1 is telling the truth, now the centralized algorithm will choose node 2 to store the 5 data packets and its utility is $\pi_2(t_1, \tilde{t}_2) = c_{V-\{2\}} - c_V + \tilde{c}_2 - c_2 = 5 - 4 + 4 - 4 = 1 > 0$, implying that node 2 has an incentive to lie. Here $c_2 = 4$ because node 2 actually stores only 2 units data packets; the other 3 units are dropped due to its energy constraint. Clearly, by reporting a lower cost type, node 2 can improve

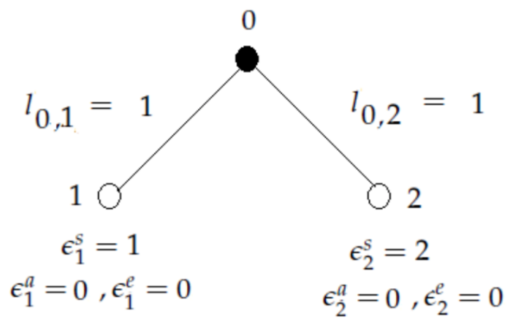

0
$l_{0,1} = 1$
$l_{0,2} = 1$
1
2
$\epsilon_1^s = 1$
$\epsilon_1^a = 0$, $\epsilon_1^e = 0$
$\epsilon_2^s = 2$
$\epsilon_2^a = 0$, $\epsilon_2^e = 0$

Figure 4: A Network Where data nodes are Energy-Constrained

its utility since it is paid according to the number of data packets it is supposed to store, yet its actual energy consumption is according to the data packets it can store under its energy constraint. The consequence of node 2's lie is that the network suffers 2 units of data loss.

Now let us examine the scenario when a node wants to exaggerate its private type. Note that by doing so, the node will never be assigned a heavier data preservation task by the centralized algorithm compared to the case of truth-telling, therefore the node will never drop any data packets. However, the system might drop some data packets when a node is critical in data preservation to the system, in the sense that the system will not be able to preserve every overflow data packet when this single node is removed from the system. We argue that a critical node might want to exaggerate its private type.

To understand why such problem could occur, note that the VCG mechanism in our baseline model works by aligning individual and system incentives. However, one discrepancy exists between their incentives when data can be dropped by individual nodes through misreporting its private type. To an individual storage node, data loss does not hurt its utility and may enhance its utility due to a lighter load of data preservation. However, to the system data loss is creating negative values. Therefore, if we continue with the standard VCG mechanism defined in our baseline payment mode, the aforementioned situation can occur that individual node misreports its private type, resulting in data loss whereas improving its own utility. The problem studied here thus shows that the application of VCG mechanism warrants scrutiny even when the system cost is a linear summation of the individual cost.

In order to restore truthfulness to the VCG mechanism and to have the cost calculation take account for data loss, we modify the VCG model in two folds: first, we add a data loss punishment to any node which drops data packets as a consequence of its lying about its private type. second, we add a data loss cost to reflect the loss in the data value when the system is infeasible and has to drop some data packets. Here we clarify more assumptions: first, no node is malice with the purpose of dropping data packets. If any data loss occurs, it is due to the limitation of the node's energy/storage capacity. Second, we assume that any data loss in the network is perfectly detectable. In addition, all data packets are equally valued and reflected by the same parameter.

Given reported type $(\tilde{t}_i, t_{-i})$, whenever node i is chosen to participate in data preservation, its payment is calculated as

$$p_i^x(\tilde{t}_i, t_{-i}) = c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - I_i\left[c_{V-\{i\}} - (\tilde{c}_V - \tilde{c}_i)\right]. \quad (19)$$

Here $I_i = 1$ if node i ever drops any data packet; $I_i = 0$ if not. Different from (17), here node i is punished by receiving zero payment when it drops any data packet. The corresponding utility of node i is thus

$$\pi_i^x(\tilde{t}_i, t_{-i}) = p_i^x(\tilde{t}_i, t_{-i}) - c_i. \quad (20)$$

Therefore, if node i drops any data packet, its utility calculated in (20) is $-c_i$, which is at most zero.

Theorem 2. *Consider a network feasible for data preservation but storage nodes are subject to energy constraints. With the* payment *given by (19) and* $t_i \in \{\epsilon_i^a, \epsilon_i^s, \epsilon_i^e\}$, *both IR and IC are satisfied.*

Proof: When there is no data loss of node i through reporting $\tilde{t}_i$, the payment is the same as in (17) and the proof follows the same as in Theorem 1. We consider the case when node i drops at least one data packet under $\tilde{t}_i$, implying that node i must be in the preservation path under $\tilde{t}_i$. There can be two cases.

Case I: Node i is not in the preservation path when reporting t_i and is in the preservation path when reporting $\tilde{t}_i$. Thus $\pi_i^x(\tilde{t}_i, t_{-i}) \leq 0 = \pi_i(t_i, t_{-i})$.

Case II: Node i is in the preservation path when reporting either t_i or $\tilde{t}_i$. The utility of i when it reports truthfully is $\pi_i(t_i, t_{-i}) = c_{V-\{i\}} - (c_V - c_i) - c_i = c_{V-\{i\}} - c_V \geq 0$. Instead when it lies by reporting $\tilde{t}_i$, its utility is $\pi_i^x(\tilde{t}_i, t_{-i}) = -c_i \leq 0$. Thus $\pi_i(t_i, t_{-i}) \geq \pi_i^x(\tilde{t}_i, t_{-i})$.
The results immediately follow.

7. SIMULATION RESULTS

Simulation Topology. Fig. 5 shows the sensor network topology that is used for our simulation experiments. 50 sensor nodes are randomly located in a field of 1000 meters by 1000 meters. The transmitting range of each sensor node is 250 meters. That is, two nodes are connected by an edge if their distance is less than or equal to the transmission range; that is, they can directly communicate with each other by sending or receiving data packets. Among the 50 sensors, nodes 0-9 are data nodes and nodes 10-49 are storage nodes. Each data node has 100 data packets, each of which has size of 512B. The storage capacity of each storage node can be varied from 26 - 50 (although 25 is the minimum storage capacity to accommodate all the 1000 data packets, as we need to take out one storage when compute utilities, we set the minimum storage capacity as 26). When it is 26, the network is almost full after all the data packets are offloaded; when it is 40, the network is exactly half-full after all the data packets are offloaded. Unless otherwise mentioned, the default values of ϵ^e, ϵ^a and ϵ^s are $100\ nJ/bit, 100\ pJ/bit/m^2$, and $100\ nJ/bit$, respectively. Below we consider infinite energy case and finite energy case, respectively. For each of the three cost parameters (ϵ^e, ϵ^a and ϵ^s), we allow a storage node to fix the value of two of them and lie about the value of the third parameter.

7.1. Infinite Energy Case

We first consider the case that each node has infinite amount of energy, thus the total energy consumption of the network can be computed using the minimum cost flow algorithms discussed in Section 4.1.

Homogeneous Case. We first study the homogeneous case wherein each node's true parameters are the same. Fig. 6 studies the half-full case and Fig. 7 the completely full case. We have several observations. First, VCG works for all the nodes, showing its truth-telling utility is never less than its lying utility. Second, $\epsilon_a$has the most dramatic effect on true vs. reported utilities compared to the effects of ϵ_s and ϵ_e, as a node's utility of lying about ε_a is more evidently different from its truth-telling utility. Second, scaling down can result in negative utility while scaling up results in at least zero utility. ThiscanbeexplainedusingEquation18: $\pi_i(\tilde{t}_i, t_{-i}) = p_i(\tilde{t}_i, t_{-i}) - c_i = c_{v-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i$ as follows.

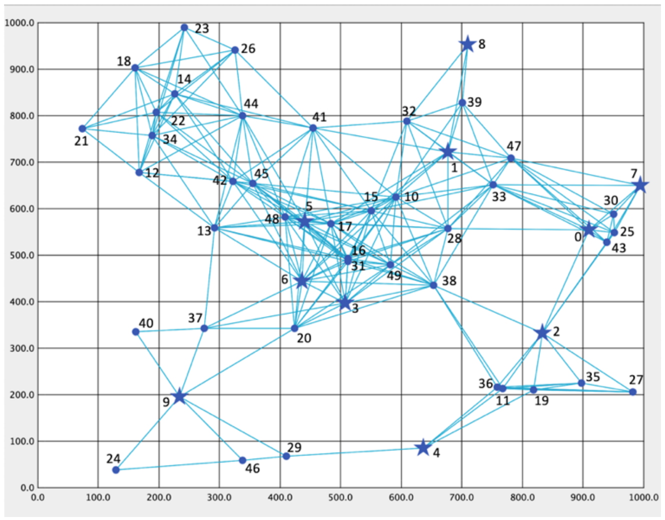


Figure 5: A sensor network with 50 nodes. Nodes 0-9 are data nodes and 10-49 are storage nodes.

When scaling down, as a node claims to cost less, its claimed cost value $\tilde{c}_i$ could get much smaller, making its total utility $\pi_i(\tilde{t}_i, t_{-i})$ negative. However, when scaling up, a node could be assigned zero tasks, making its utility zero. Third, when the network scenario gets completely full and more stressful, Fig. 7(b) shows that more lying nodes gets negative utility. This is because with completely full storage, it is more likely that lying storage node is assigned a heavy load, and as such $\tilde{c}_i - c_i$ becomes negative.

Fig. 6 illustrates the true vs. reported utilities in the half-full homogeneous case when nodes lie about their ϵ^s parameter. It also illustrates how the reported utilities change as ϵ^s is scaled by different scaling factor α. For example, Node 17 acquires the same amount of utility when α =0.01, 0.1, and 1.0. 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.

<u>Heterogeneous Case</u>. We then study the heterogeneous case wherein different nodes' true parameters could be different. Fig. 8 studies the half-full case and Fig. 9 the completely full case. We have the same observations as in homogeneous case.

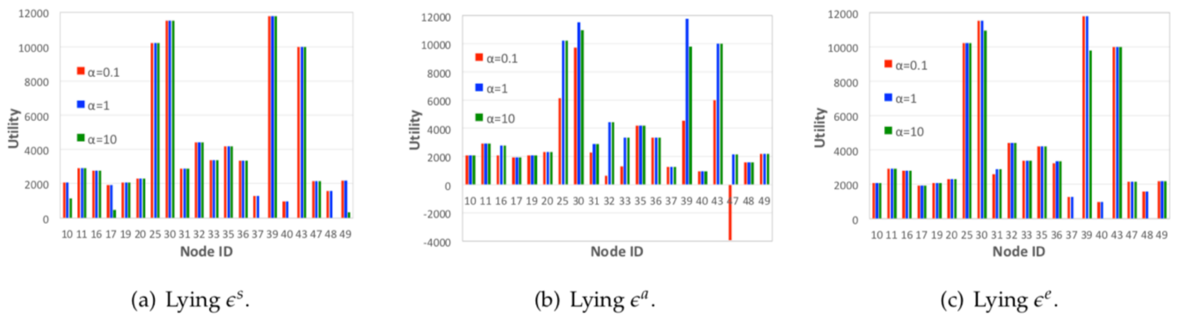
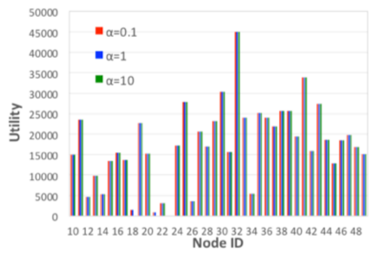
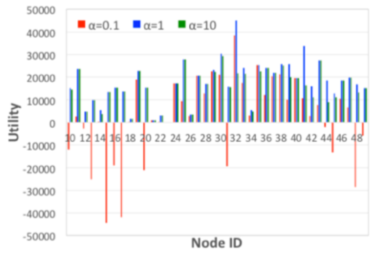


Figure 6: Homogeneous parameters in half full network.


(a) Lying ϵ^s.


(b) Lying ϵ^a.

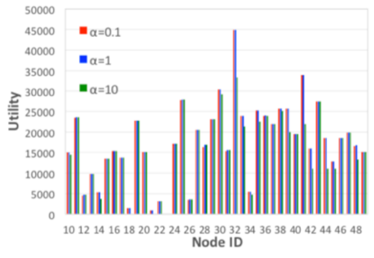
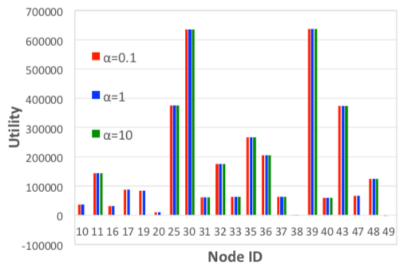
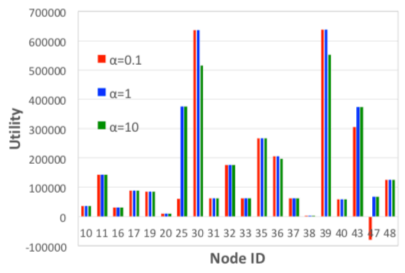

(c) Lying ϵ^e.

Figure 7: Homogeneous parameters in completely full network.


(a) Lying ϵ^s.


(b) Lying ϵ^a.

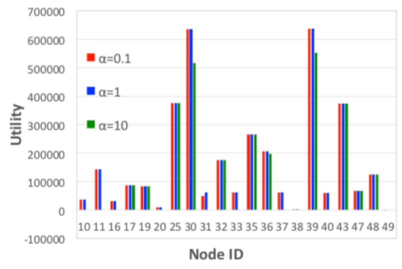
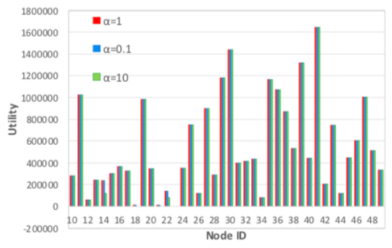
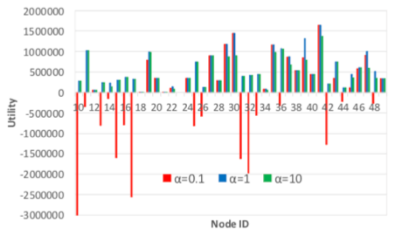

(c) Lying ϵ^e.

Figure 8: Hetergeneous parameters in half full network.


(a) Lying ϵ^s.


(b) Lying ϵ^a.

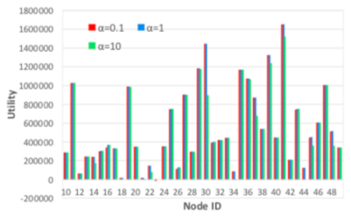

(c) Lying ϵ^e.

Figure 9: Hetergenous parameters in completely full network.

Lying With Three Parameters. Next, we let each node to lie about all of its three parametersϵ_a, ϵ_e, and ϵ_s. That is, each node randomly and independently chooses a value in the range of [0.1, 10]. Fig. 10 shows comparison of node's utilities between truth-telling and lying. It shows that truth-telling is always the dominate strategy and our VCG theory works for the general case of lying of multiple parameters.

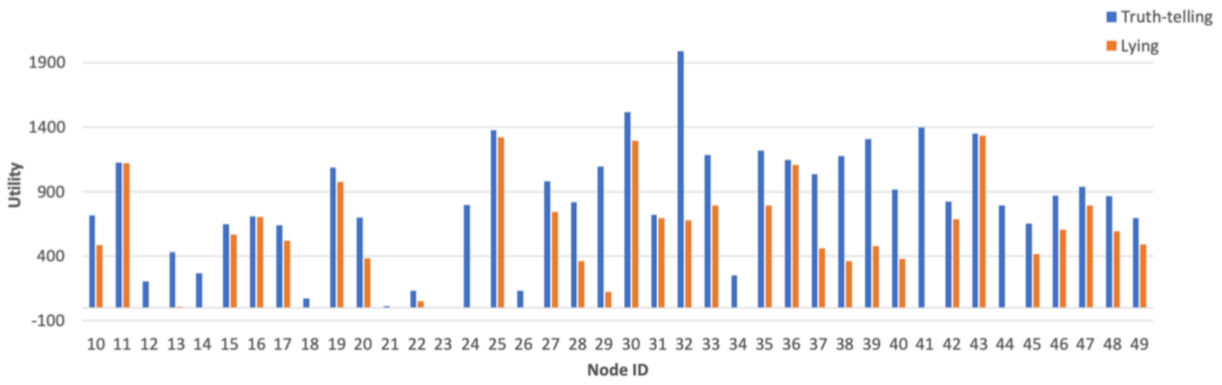


Figure 10: Utilities of nodes when lying all of its three parameters simultaneously.

7.2. Finite Energy Case

In this part we conduct three phases of investigation in a progressive manner. In Section 7.2.1, we focus on the centralized data preservation algorithms with truth-telling and study its network characteristics w.r.t. initial energy levels and storage capacities of sensor nodes. In particular, we are interested in the workload of each node (that is, number of data received, transmitted, or saved data packets), its energy consumption, as well as the dead nodes (i.e., the nodes that deplete their energies during the data preservation process). Second, in Section 7.2.2, we study the system wherein each node one by one lies about their data preservation parameters. We check how VCG works by comparing each node's lying utility with its truth-telling utility. We find that there is one node that does not follow the VCG theory although the rest do. Third, in Section 7.2.3, we check the detail network characteristics in VCG. Fourth, in Section 7.2.4, we focus on the ab- normal node identified in Section 7.2.2 and study its network behavior thoroughly under different network parameters and identifies the cause. Finally implement our fix and show it indeed works in Section 7.2.5.

7.2.1. **Network Characteristics in Truth-Telling.**

Minimum Feasible Initial Energy Level. We define *minimum feasible energy level* as the minimum energy level at which all the 1000 data packets can still be offloaded into the network shown in Fig. 5. We denote it as E_m mJ; that is, when each node's initial energy level is $(E_m - 1)$ mJ, the maximum amount of data packets offloaded is less than 1000.

To find E_m given any sensor network instance, we decrease the initial energy level of sensor nodes from 1600 mJ and record the number of dead nodes and number of data packets offloaded at different levels. At each energy level, we use the maximum flow LP proposed in Section 4.2.1 to find out the maximum number of packets are offloaded, then we use the minimum cost flow LP proposed in Section 4.2.2 to find out the minimum total energy consumption offloading that number of data packets.

To find dead nodes, we analyze data preservation paths resulted from the minimum cost flow LP to calculate the energy consumption of each node. A data node is dead if it does not have enough energy to relay (i.e., receive then transmit) at least one data packet to its closest neighbor; that is, its remaining energy is less than the energy to relay one packet to its closest neighbor. For a storage node, we compute its energy cost of saving one data packet to its storage and its energy cost of relaying one data packet to its closest neighbor. A storage node is dead if its remaining energy is less than the smaller one of above two energy costs.

Fig. 11 shows that the number of dead nodes increases from 0 at 1600 mJ to 4 at 1312 mJ, while the network is still able to offload all the 1000 data packets. It also shows the IDs of the dead nodes at each energy level. Among the four dead nodes, three are data nodes (i.e., nodes 0, 2, and 4) and one is storage node (i.e., node 32). When we further decrease it to 1311, 999 data packets can be offloaded. We thus set E_m as 1312 mJ for the rest of simulation unless otherwise mentioned.

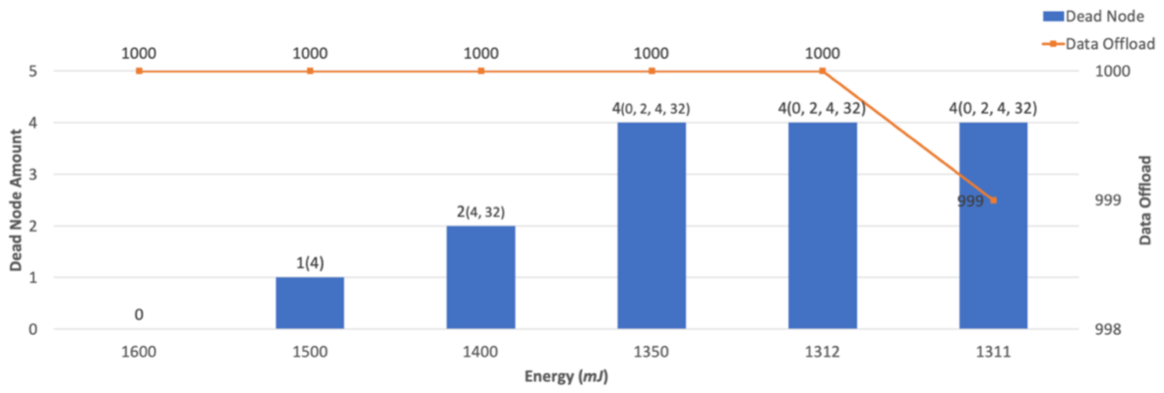


Figure 11: Number of dead nodes and offloaded data packets w.r.t. initial energy levels.

Work Load and Energy Consumption of Each Node. Fig. 12 shows the workload and energy consumption of individual nodes. It shows that node 48 has the most work load, with the most number of received packets and transmitted packets. This is because node 48 is close to data nodes 3, 5, and 6, thus serves as the "traffic hub" to offload their data packets to the storage nodes located at the top left region of the sensor field in Fig. 5. However, although it is most work loaded, its energy consumption is second to the one of node 32, which has already depleted its 1312 mJ of energy. Like node 48, node 32 is also close to a few data nodes (i.e., node 1 an 8); however, unlike node 48, node 32 is relatively distant to neighboring storage nodes, thus costing more transmission energy to relay node 1 and 8's data packets to other storage nodes.

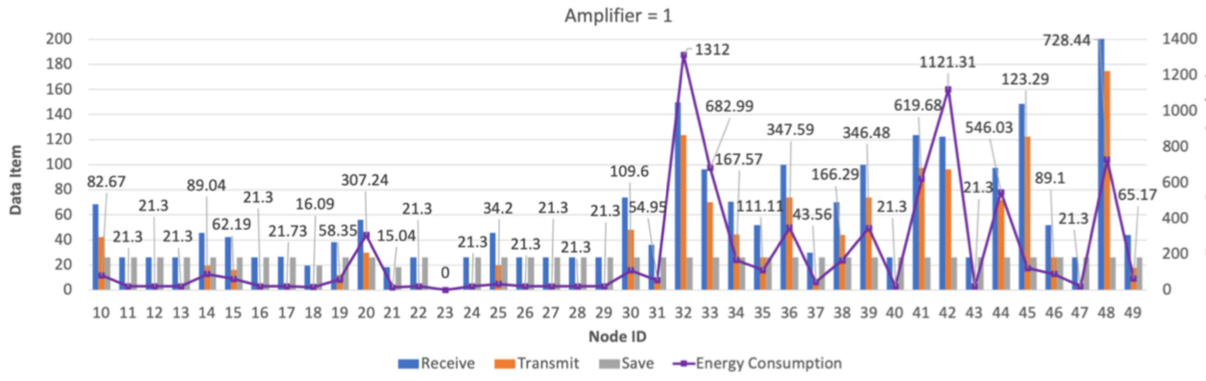


Figure 12: Workload and energy consumption of each node when truth-telling.

7.2.2. **VCG Performance of Nodes.**

In this part, we investigate how cheat-proof VCG works in our network. In particular, each storage node one by one lies about each of the three parameters viz. ϵ^a, ϵ^e, and ϵ^s, and computes its lying utilities and compares with its truth-telling utility. We set the initial energy level of nodes as 1312 mJ and their storage capacity as 26 at which the network is almost full after data offloading.

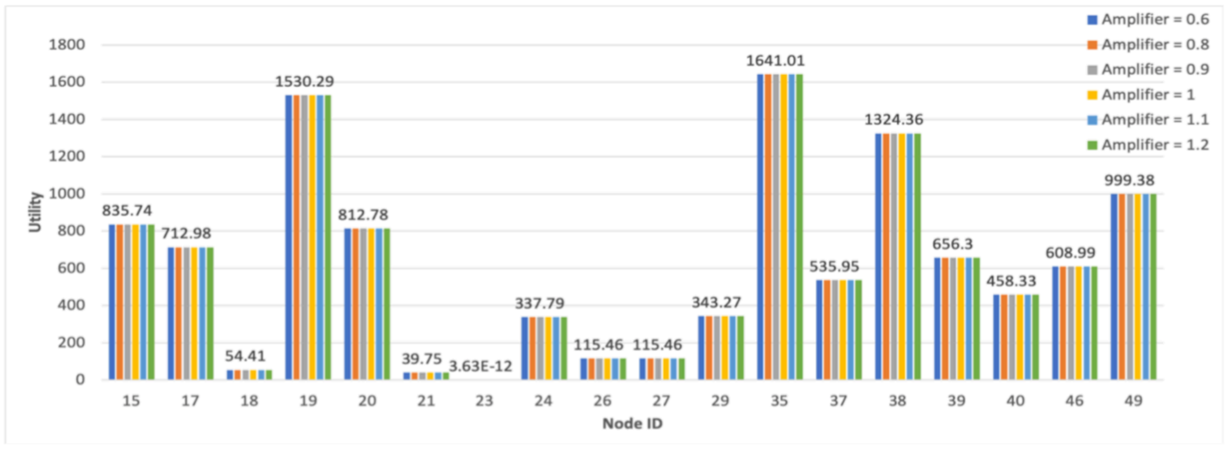


Figure 13: Nodes with same truth-telling and lying utilities when varying ϵ^a.

Varying ϵ^a. We vary ϵ^a from 0.6, 0.8. 0.9, 1, 1.1, to 1.2. Fig 13 shows that out of 40 storage nodes (i.e., node 10 to 49), 18 of them have the same values of truth-telling and lying utilities. This is consistent to the VCG theory that the truth-telling utility is never less than the lying utility. Moreover, the reason they have the same truth-telling and lying utilities is because they participate in the same way whether lies or not; i.e., each such node receives, transmits, and saves the same amount of data packets. We note that node 23's utilities are all zeros for truth-telling or lying. As 23 is farthest from all the data nodes in the sensor field in Fig 5, it does not participate at all in the minimum-energy data preservation process, incurring zero payment, cost, and utility.

Fig. 14 shows the rest of 22 storage nodes with at least one lying utility different from its truth-telling utility. We observe that while 21 of them comply with our VCG theory, node 32 does not as its truth-telling utility of 2014 being less than its lying utilities at scaling factors of 0.6, 0.8, and 0.9, which are 2024, 2043, and 2054, respectively. This can be explained using Equation 18: $\pi_i(\tilde{t}_i, t_{-i}) = p_i(\tilde{t}_i, t_{-i}) - c_i = c_{v-\{i\}} - (\tilde{c}_V - \tilde{c}_i) - c_i$, as follows. As the first term of its r.h.s, which is $c_{v-\{i\}}$, is the total energy consumption excluding i, it does not change whether i lies or not. Its second term, $\tilde{c}_V - \tilde{c}_i$, is all other nodes' total energy consumption excluding i. When i scales down its factor by claiming it costs less energy than it actually does, it receives more data packets

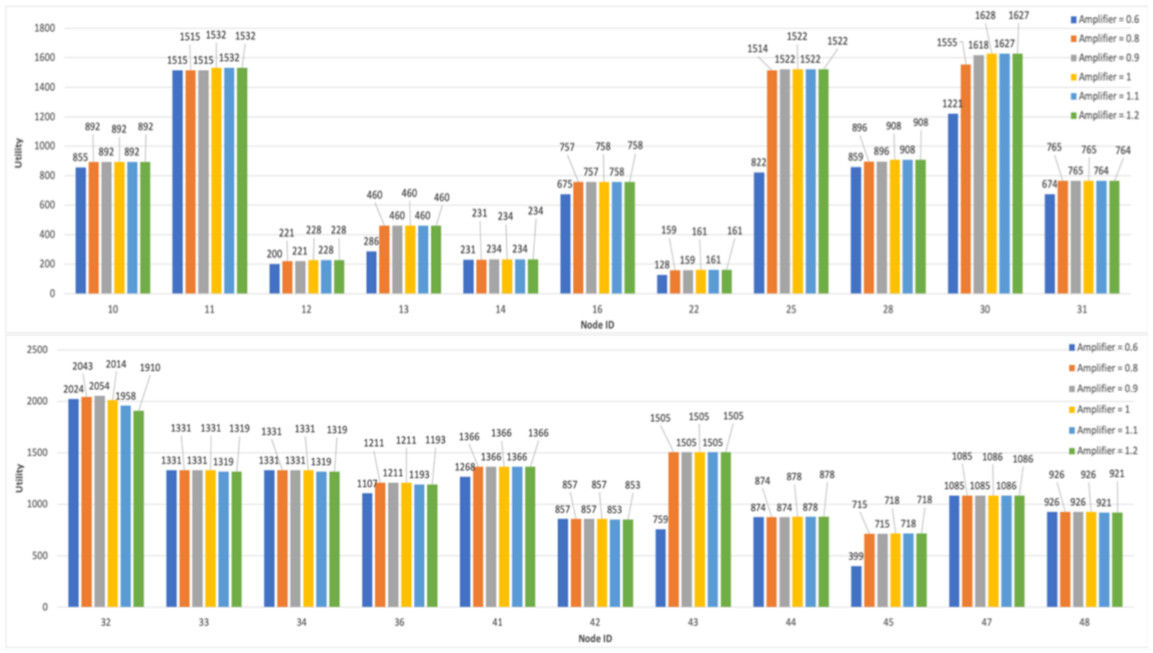


Figure 14: Nodes with different truth-telling and lying utilities when varying ϵ^a.

than what its energy allows and consequently, other nodes's total energy consumption $\tilde{c}_V - \tilde{c}_i$ gets less. The third term c_i is node $i's$ real cost no matter it lies or not, which is restricted by $i's$ initial energy level, thus is around the same whether i lies or not. Therefore, i's utility $\pi_i(\tilde{t}_i, t_{-i})$ could gets larger when it scales down, which explains node 32's abnormal behavior.

Note that when node 32's scaling factor is 1.1 and 1.2, its lying utility is smaller than its truth- telling utility, which is consistent with the VCG prediction. As node exaggerate its parameter and claims to cost more energy than necessary when performing a task, the system passes less number of packets to it and pays it less, resulting in less utility compared to when truth-telling.

Varying ϵ^e We then let each node lie about its receiving amplifier ϵ^e and compare its lying utility with its truth-telling utility. Fig. 15 shows that out of 40 storage nodes, 29 of them have exactly the same truth-telling and lying utilities. Compare to Fig 13 of varying ϵ^a, which shows 18 storage nodes with the same truth-telling and lying utilities, varying ϵ^e has less impact on nodes' utilities.

Fig. 16 shows the rest 11 storage nodes that have different truth-telling and lying utilities when varying ϵ^e. Again, node 32 is the only node that violates the VCG theory by showing that its truth- telling utility of 2014 is less than its scaling-down utility at factor 0.6, which is 2016. However, the difference between these two is just 2, which is much smaller than the that in Fig. 14, which

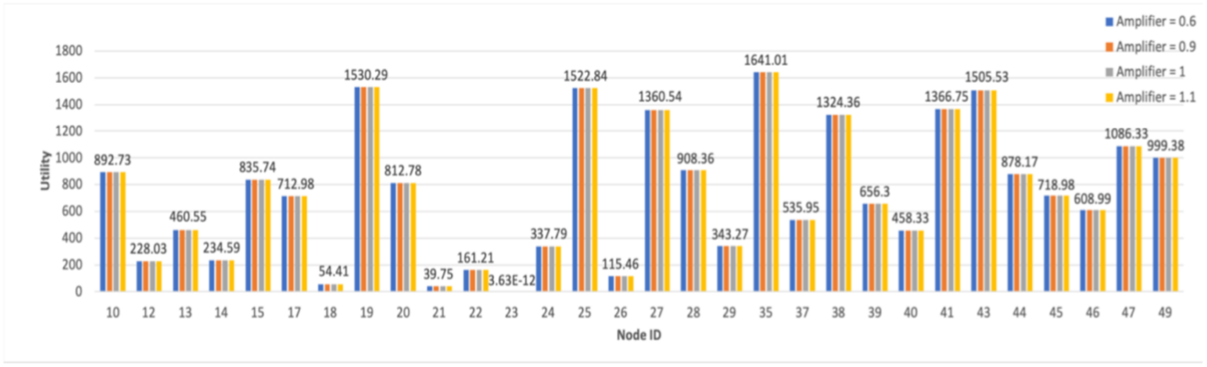


Figure 15: Nodes with same truth-telling and lying utilities when varying ϵ^e.

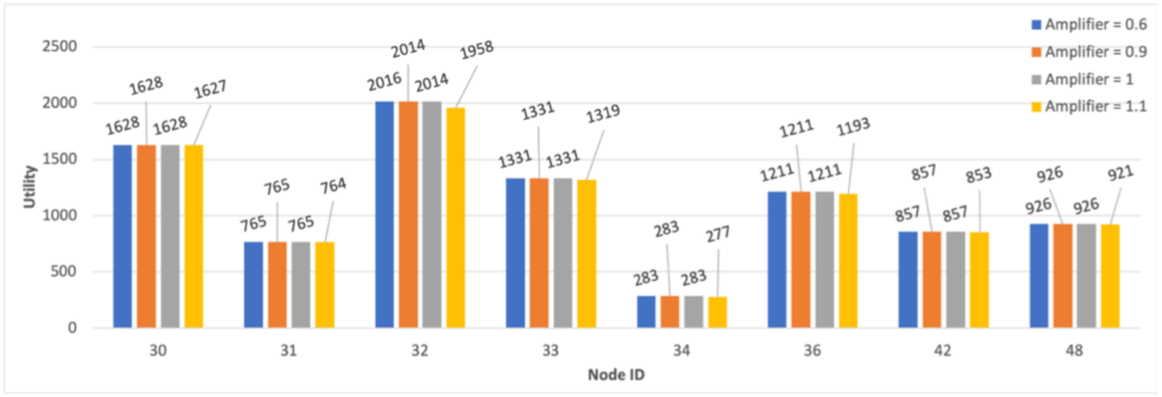


Figure 16: Nodes with different truth-telling and lying utilities when varying ϵ^e.

has the maximum difference of 40. It shows again for a storage node, lying about ϵ^e has less effect on its utility compared to lying about ϵ^a.

Varying ϵ^s. Finally, we compare the truth-telling utility of each storage node with its utility when lying about its saving amplifier ϵ^s. Fig. 17 shows the utilities of the 32 storage nodes with the same values of truth-telling and lying utilities while Fig. 18 shows the utilities of the rest 8 storage nodes with different truth-telling and lying utility values. Like ϵ^e, ϵ^s also has a smaller effect upon a lying node's utility compared to ϵ_a.

Total Energy Consumption of the Network Under Lying. Fig. 19 shows the total energy consumption of data preservation of the entire network when each node lies about its ϵ_a. It shows that in most of the cases, total energy consumption when truth-telling is no larger than that of when lying, demonstrating the optimality of our minimum cost flow ILP proposed in Section 4.2.2.

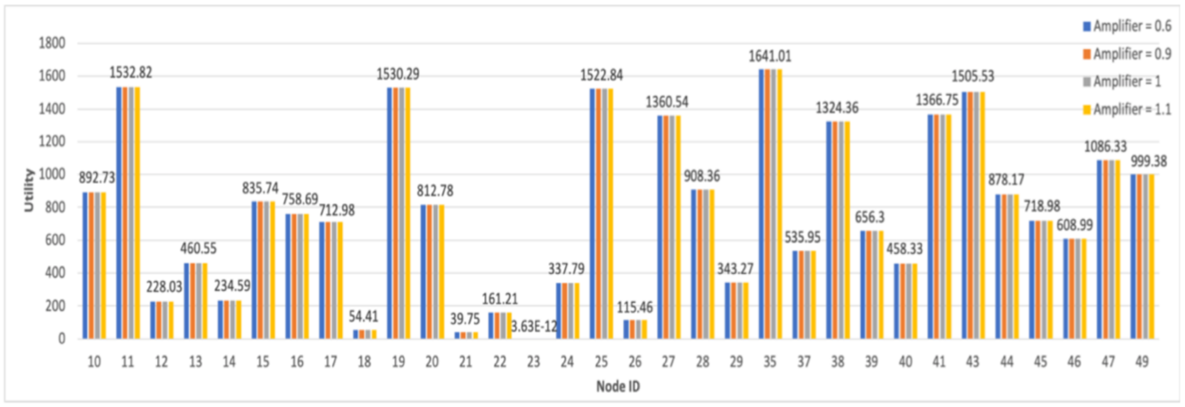


Figure 17: Nodes with same truth-telling and lying utilities when varying ϵ^s.

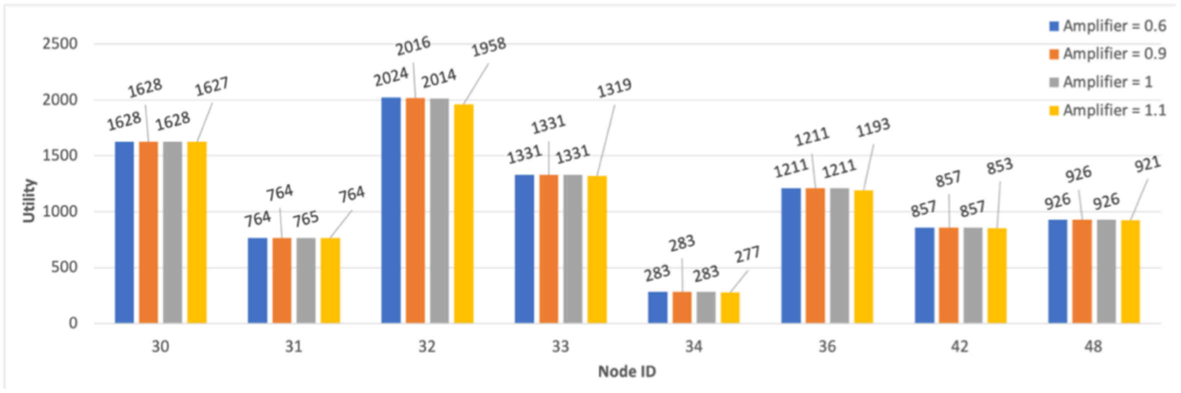


Figure 18: Nodes with different truth-telling and lying utilities when varying ϵ^s.

However, we also observe that when a few nodes lie at $\epsilon_a = 0.6$, the resulted total energy consumption is much larger than the optimal energy consumption. This is because when they claim they are very efficient in data preservation, the system assigns them lots of data packets that are not assigned in optimal solution via those non-optimal data preservation paths, resulting in much larger energy consumption.

7.2.3. **Network Characteristics under VCG.**

Next we investigate the effect of VCG by finding each node's network characteristics when lying. Such network characteristics of a node include its workload, its discarded data packets, and its energy consumption. Here, the workload of a storage node includes three parts: the number

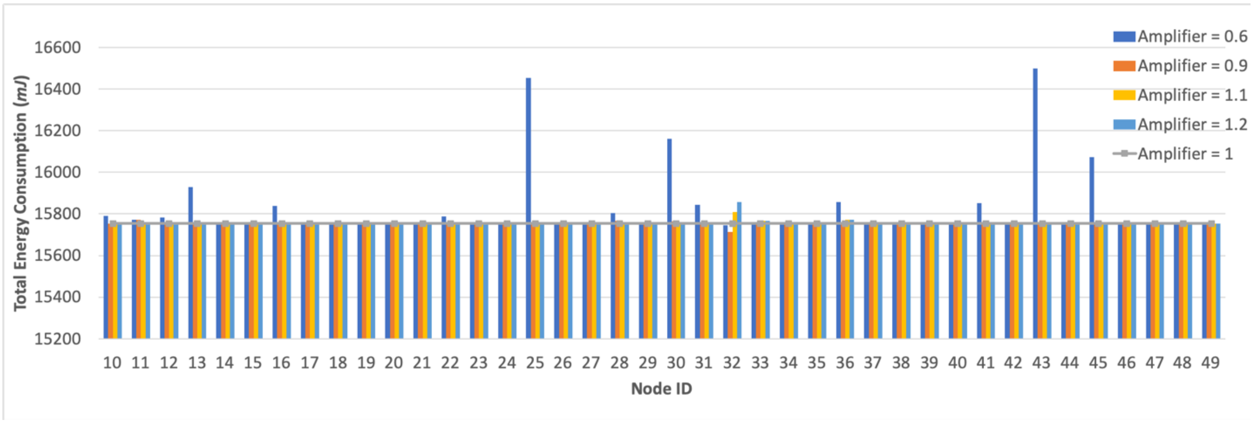


Figure 19: Comparing total energy consumption in truth-telling and lying.

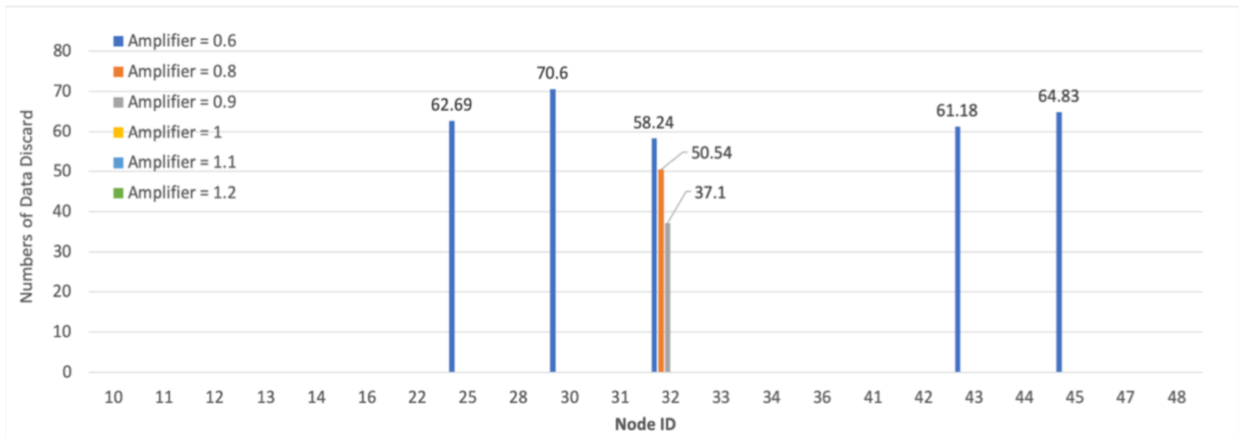


Figure 20: Nodes that discard data packets when lying with different amplifiers.

of data packets it receives, the number of data packets it saves, and the number of data packets it transmits. The workload of a storage node shows its "contribution" to the data offloading process. The number of discarded data packets is referred to as *data loss*. This number is obtained by subtracting the total number of packets a node saves and transmits from the total number of packets it receives from other nodes. As changing ϵ^a has more impact upon node's utility, we adopt ϵ^a as the parameter that each node lies about. Fig. 20 shows the nodes that discard data packets when lying with different scaling factors of ϵ^a while fixing storage capacity as 26 and initial energy level as 1312 mJ. It shows that among the 40 storage nodes, only five storage nodes viz. nodes 25, 30, 32, 43, 45 discard data in at least one of the three amplifiers viz. 0.6, 0.8, 0.9. The topology in Fig. 5 shows that all these nodes are close to one or more data nodes, thus participating heavily and possibly consuming lots of energy in transmitting data packets for the data nodes.

To find out why there are data loss, we present the detailed analysis of the workload and energy consumption of each node for amplifier = 0.6, 0.9. and 1.1, respectively, as shown in Fig. 21, 22, and 23. For example, Fig. 21 shows that at amplifier 0.6, all the five nodes 25, 30, 32, 43, 45 all have energy consumption of 1312 mJ, depleting their energy power. Fig. 22 shows that at amplifier 0.9, node 32 has energy consumption of 1312 mJ, depleting its energy power. It becomes clear that those nodes discard data packets are due to that they deplete their energy power. When amplifier is 1.1 or 1.2, none of the nodes deplete their energy, thus no data discarding in these cases.

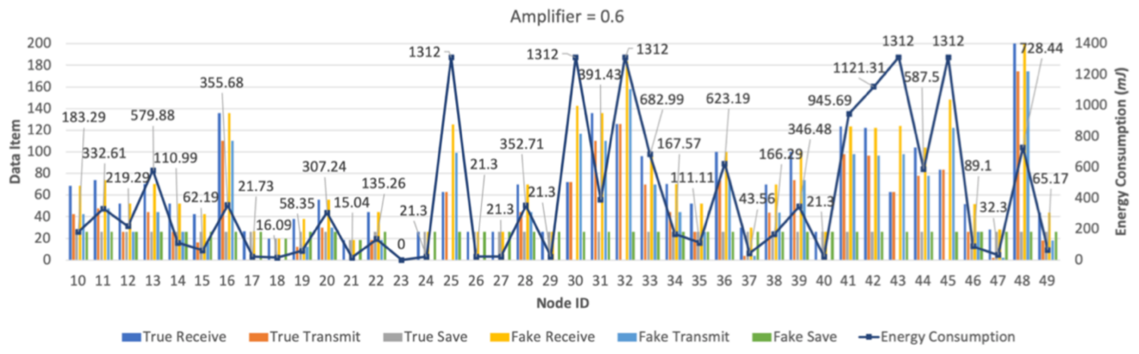


Figure 21: Workload and energy consumption of each node when amplifier = 0.6.

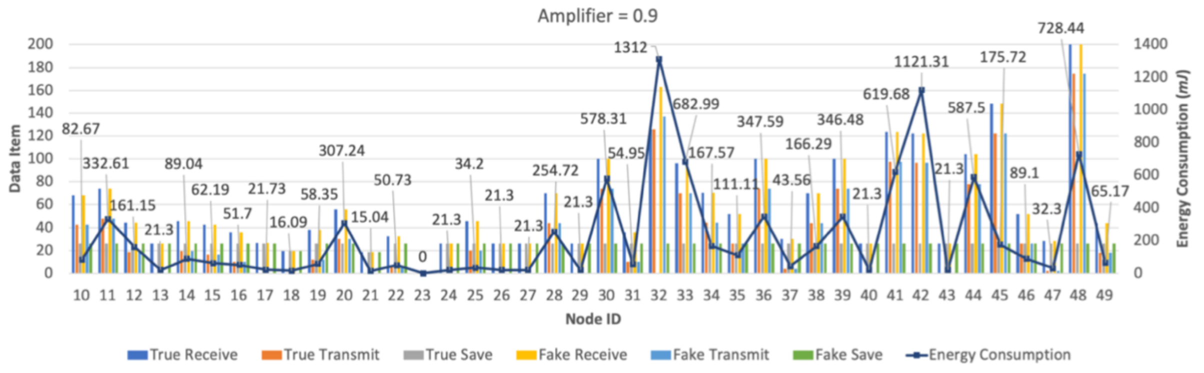


Figure 22: Workload and energy consumption of each node when amplifier = 0.9.

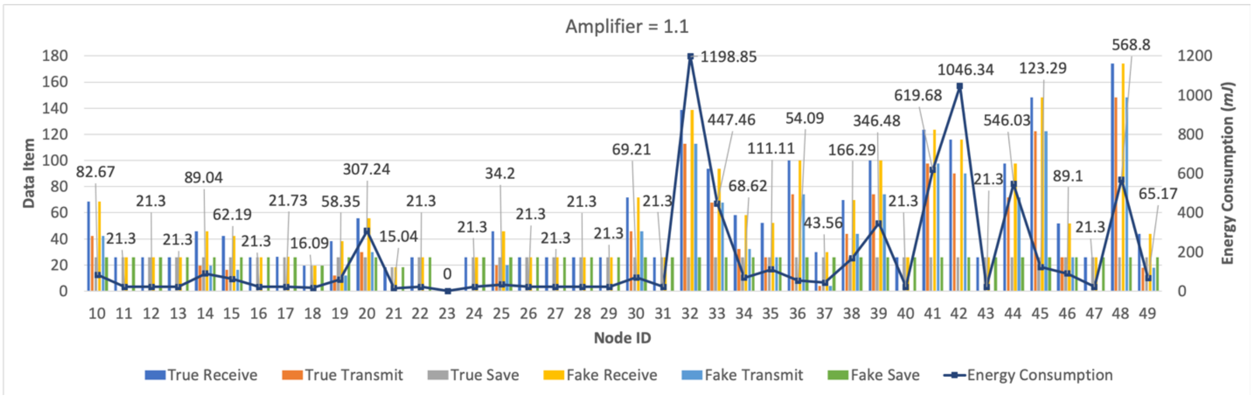


Figure 23: Workload and energy consumption of each node when amplifier = 1.1.

7.2.4. **Investigating Node 32.**

As Section 7.2.2 shows that node 32 has incentive to lie when VCG is applied, next we explain why VCG could fail to yield truth-telling in the data-preservation problem. As varying ε_a has more influence upon a node's utility calculation, we let node 32 to lie about its ε_a only. In this section we are trying to answer the following question: among all the 40 storage nodes, why only node 32 shows behavior that is not consistent with VCG? We try to answer this question from two quantities of each individual sensor node: its data loss and its utilities under both lying and truth-telling.

Investigating Data Loss of Node 32. We take a close look of how many data packets node 32 discards. Fig. 24 shows the data loss of node 32 while increasing the initial energy levels of sensor nodes; and at each energy level, we vary the scaling factors from 0.6, 0.7, 0.8, 0.9, to 1.0. We have three observations. First, at each energy level, with the increase of scaling factor, the data loss decreases. This is because as the scaling factor gets close to 1, which is the truth-telling case, node 32's extend of lying gets less. As such, the number of data packets sent to node 32 becomes smaller, making it discarding less number of data packets. Second, for the same scaling factor, with the increase of energy level, the data loss decreases too. This is because with the increase of energy level, node 32 is able to deal with more data packets, thus discarding less number of data packets. Our final observation is that when the energy level is high (at 1500 mJ and 1550 mJ) and when the amplifier is closer to 1 (at 0.8 and 0.9), there is no data loss and node 32's lying behavior is compensated by its enough energy. Note that there is no data loss at all at scaling factor of 1.0 for node 32 in all energy levels; as it is truth-telling, it always receives the amount of data it can deal with thus it does not discard any data.

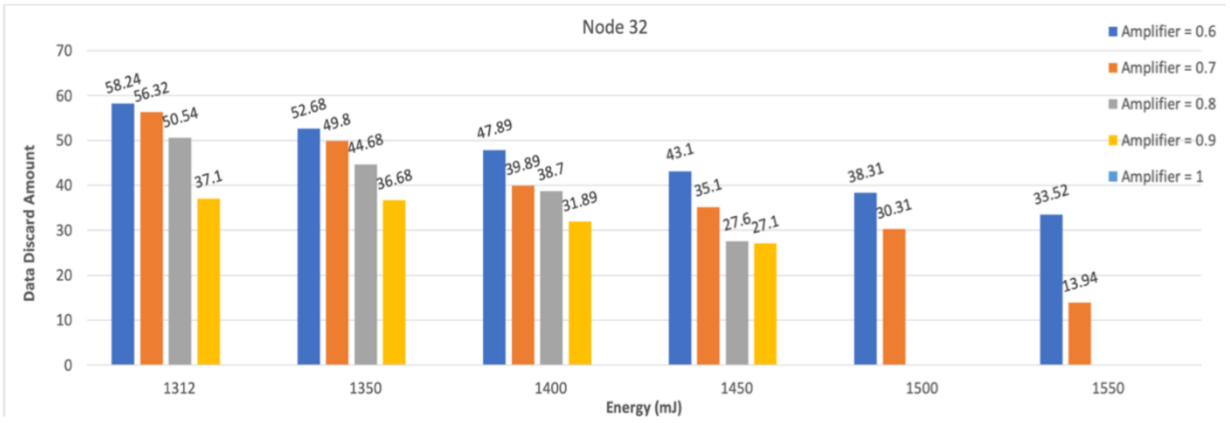


Figure 24: Number of discarded data at node 32 when varying its initial energy.

Fig. 25 shows node 32's data loss when we change its storage capacity at different amplifier while fixing the initial energy as 1312 mJ. It shows that the data loss decreases as the storage capacity increases, as node 32 have more space to store data thus discarding less data. We again observe that with the increase of the amplifier, node 32 discards less amount of data, as it behaves more and more truthfully. When amplifier is 1.1, there is no data loss at all. As node 32 claims to spend more energy that it is necessary, it receives less data than in the truthful case thus has enough energy to process all such received data.

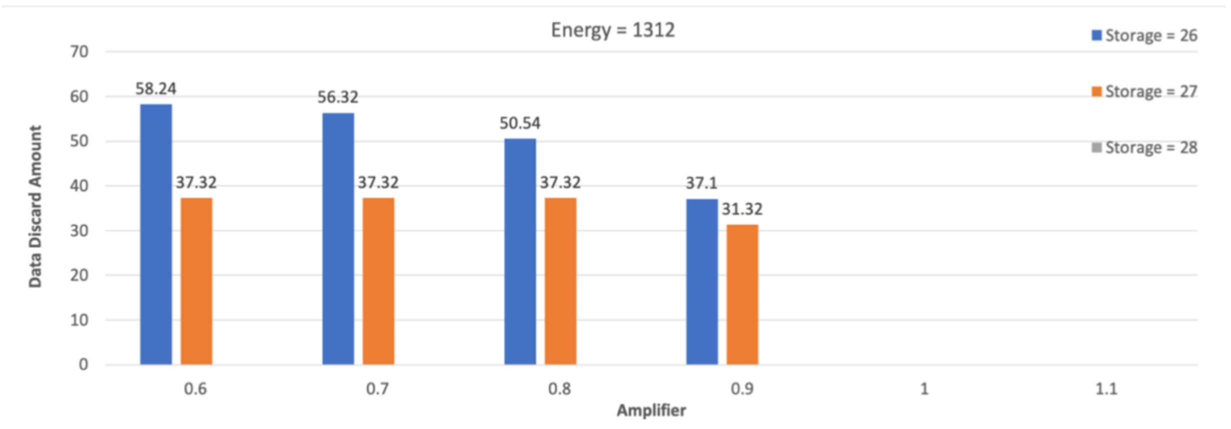


Figure 25: Number of discarded data at node 32 when varying its storage capacity.

Investigating Utility of Node 32 by Varying Energy. Fig. 26 shows the utility of node 32 when increasing the initial energy level from 1312 mJ to 1550 mJ while varying ϵ_a and fixing the storage as 26 at each energy level. We find that when the initial energy level is under or at 1450 mJ, node 32 shows abnormal that its truth-telling utility is smaller than at least one of its lying utilities. However, when the energy level is greater than or equal 1500 mJ, node 32 follows the AMD theory in which the truth-telling utility is the dominate strategy. This can be explained using incentive to lie defined below.

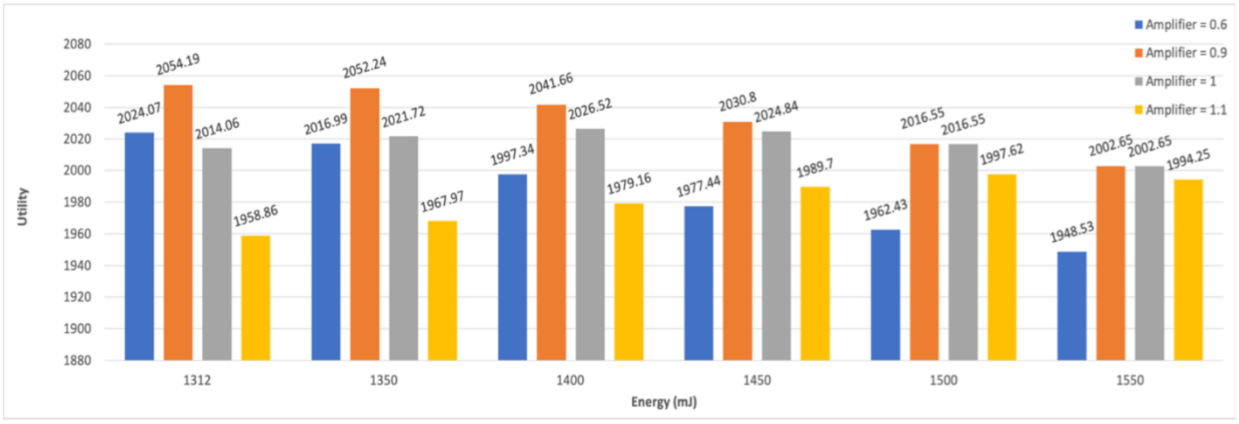


Figure 26: Utility of Node 32 when varying initial energy level.

Investigating Incentive to Lie of Node 32. For any storage node, its *incentive to lie* is its lying utility minus its truth-telling utility; the more utility it gets when lying the larger of its incentive to lie. We re-plot Fig 26 as Fig 27 to show the incentive to lie for node 32 under different initial energy levels. When amplifier is less than 1 (i.e., 0.6 and 0.9), its incentive to lie decreases with the increase of the initial energy levels. This is because data loss by node 32 eventually drops as system energy increases (see Figure 24), shrinking the potential benefit accrued to node 32 through its data loss. As a result, the incentive for node 32 to lie also drops.

When amplifier is greater than 1, the incentive to lie is always negative based on our assumption that no single node is critical to the system feasibility, therefore no data loss could occur for amplifier is greater than 1. Moreover, the curve in the figure goes up and closer to zero, showing that node 32's utility loss under lying drops when energy constraint is more relaxed. The reason is that node 32's lie becomes less harmful to the system performance when the energy constraint is more relaxed, and the job load assigned to node 32 gets less and less differentiated from its job load under truth-telling. Indeed, if under certain energy level node 32 is assigned the same task under lying with 1.1 amplifier and under truth-telling, its incentive to lie becomes zero.

Investigating Utility When Varying Storage Capacity. Next we fix the energy as 1312 mJ and

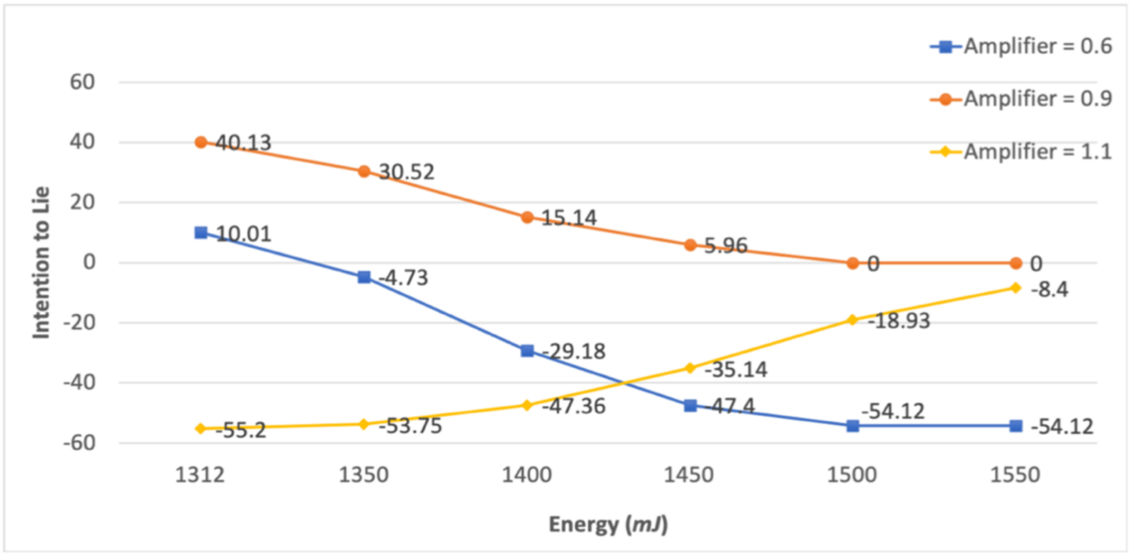


Figure 7: Incentive to lie of node 32 when varying initial energy level.

compare node 32’s utilities by varying storage capacity of sensor nodes. Fig 28 shows that when amplifiers are 0.6 and 0.9 while storage capacities are 26 and 27, truth-telling utilities are less than the lying utilities, which is against the AMD theory. However, when storage capacity gets to 28, the truth-telling utility will be larger than lying utility. This shows that when storage capacity of sensor nodes are small, nodes have more incentive to lie in order to gain more utilities, which is further explained in Fig. 29 below.

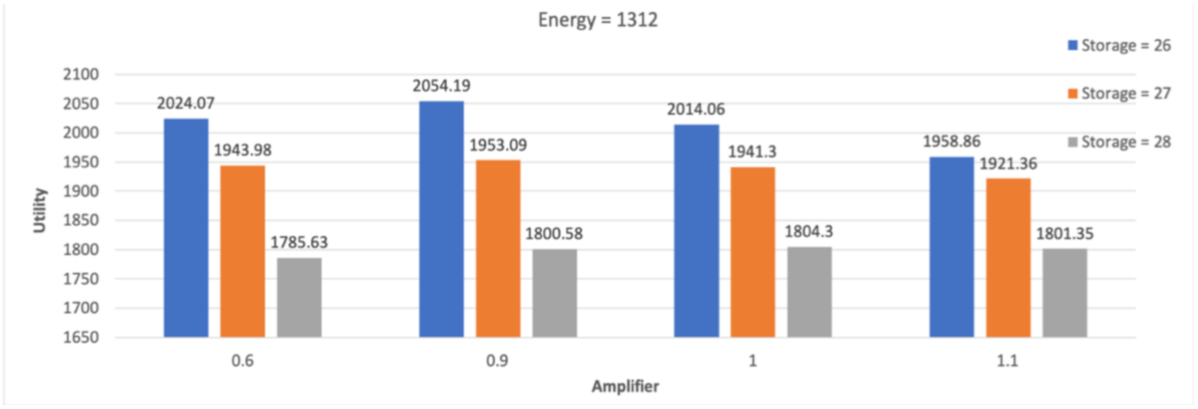


Figure 28: Utility of node 32 when varying storage capacity.

<u>Investigating Incentive to Lie of Nodes When Varying Storage.</u> Fig. 29 investigates the incentive to lie for node 32 under different storage capacities. When the amplifier is smaller than 1 (i.e., 0.6 and 0.9), its intention to lie is decreasing when increasing the storage capacity. The main reason is that under a larger sensor network storage capacity, each node does not need much energy to offload data. So, lying results in less distortion in the data preservation route, meaning that data loss under lying drops and the intention to lie will decrease. We also observe while amplifier is equal to 1.1 and we increase the storage capacity, the difference between lying utility and truth-telling utility will decrease.

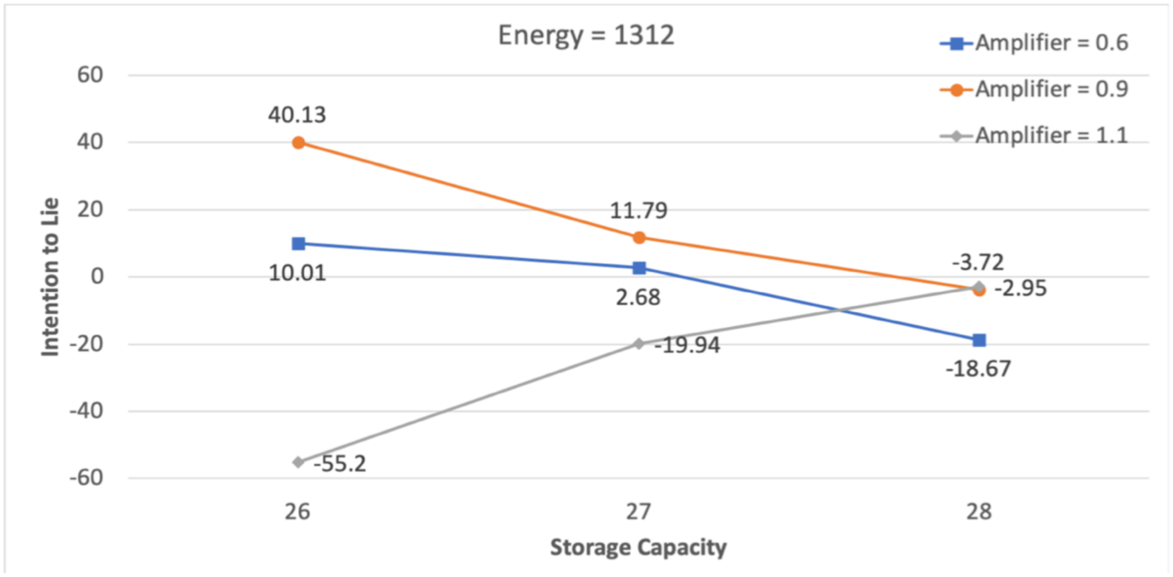


Figure 29: Incentive to lie of node 32 when varying storage capacity.

7.2.5. **Modified VCG Mechanism for Truth-Telling.**

As node can lie about its parameters in order to gain more utility, we modify the VCG mechanism to fix this problem. Our fix is simple: for any node that drops its received packages, the system gives it zero payment. Dropping packets can be observed as wireless communication is broad- cast, any node's neighbors knows exactly how many data packets it transmits. As the number of received packets minus the number of stored packets is the number of transmitted packets while each node's storage capacity is a public knowledge, whether a node drops received data packets or not thus can be easily observed.

The we implement above data loss inhibiting mechanism and calculates the utilities of each node when it is truth-telling or lies with different amplifiers. Fig. 30 and 31 shows the utilities of nodes with the data loss inhibiting mechanism implemented. With this mechanism, for each node, the truth-telling again is the dominating strategy; that is, the truth-telling utility of each node is always greater than or equal to its lying utilities.

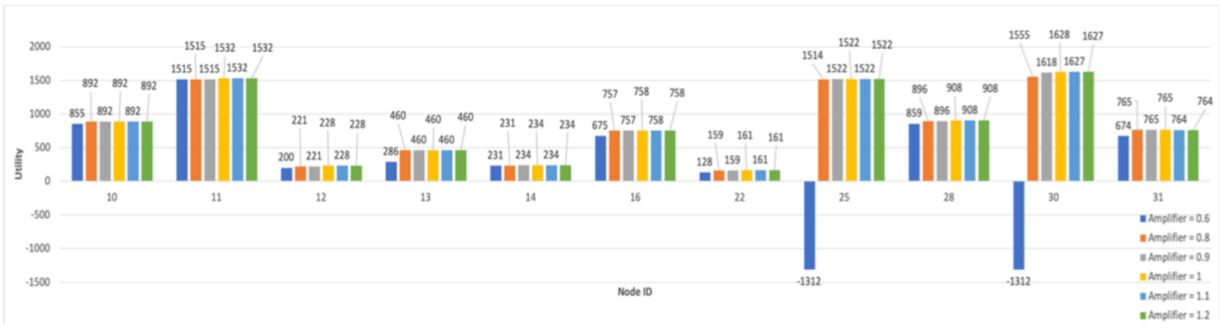


Figure 30: Utilities of nodes in Modified VCG.

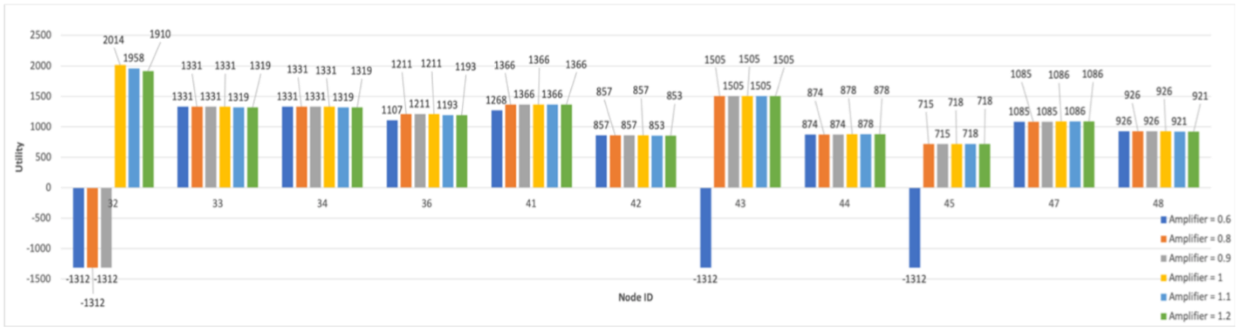


Figure 31: Utilities of nodes in Modified VCG.

8. CONCLUSIONS AND FUTURE WORK

In this work, we study data preservation problem in base station-less sensor networks wherein energy- and storage-constrained sensor nodes behave selfishly. We take a game theoretic approach and design a payment model under which the individual sensor nodes, motivated solely by self- interest, achieve good system-wide data preservation solution. In particular, we break down the data preservation cost of each storage node into two parts: relaying cost and storing cost, where cost parameters are node-dependent. The payment model is designed in a way such that no matter which cost parameter (related only to the relaying cost or only to the storing cost or to both) is private to the node, truthfully reporting the cost parameter is a dominant strategy to each node. We show that as a result, in the game it is an equilibrium that each storage node first truthfully reports its cost parameter, then participates in data preservation if it is chosen by the centralized data preservation algorithm.

In the next step of the work, we will validate theoretical findings using simulation results. By contrasting the payment of each storage node in the sensor network under truth-telling strategy to what it is under lying, we will show that truth-telling is never worse off and in certain cases is strictly better off to each storage node regardless of the choice of the other nodes. The simulation results thus can verify that truth-telling is a dominant strategy of each data node. Other future work includes relaxing some assumptions in the current work. In particular, we have assumed that data preservation is feasible in the sensor network, which implies two assumptions: first, the total size of the overflow data packets can be accommodated by the total available storage spaces in the network. Second, there is no energy constraint of each node, i.e., all the nodes have enough energy to offload and preserve all the overflow data packets. If any of the two assumptions is re- laxed, the network is infeasible and some data packets may inevitably be lost. If so, it is interesting to see how the payment model can work to induce the efficient data preservation. Finally, we will extend our analysis to a dynamic scenario wherein overflow data are generated from time to time at different nodes. It is well understood in game theory that an infinitely repeated game gives a much larger set of equilibrium and in certain scenarios full cooperation can be achieved. In our setting of data preservation among selfish nodes, it is interesting to see to what extent we need to provide motivation for selfish storage node to engage in the optimal data preservation.

9. REFERENCES

[1] *Andrewgoldberg'snetworkoptimizationlibrary*.http://www.avglab.com/andrew/soft.html.

[2] G. Aathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Ultra-low power data storage for sensor *networks*. *ACM Trans. Sen. Netw.*, 5(4):33:1–33:34, 2009.

[3] R.K.*Ahuja*,T.L.Magnanti,andJ.B.Orlin.*NetworkFlows:Theory,Algorithms,andApplications*. Prentice Hall, Inc., 1993.

[4] Tarek AlSkaif, Manel Guerrero Zapata, and Boris Bellalta. Game theory for energy efficiency in wireless sensor networks: Latest trends. *Journal of Network and Computer Applications*, 54:33 – 61, 2015.

[5] A. Cammarano, C. Petrioli, and D. Spenza. Pro-energy: A novel energy prediction model for solar *and* wind energy-harvesting wireless sensor networks. In *IEEE 9th International Conference on Mobile Adhoc and Sensor Systems (MASS 2012)*.

[6] Dimitris E. Charilas and Athanasios D. Panagopoulos. A survey on game theory applications in *wireless* networks. *Comput. Netw.*, 54(18):3421–3430, December 2010.

[7] E. H. *Clarke*. Multipart pricing of public goods. *Public Choice*, 1971.

[8] E. Cochran, J. Lawrence, C. Christensen, and A. Chung. A novel strong-motion seismic net-work *for* community participation in earthquake monitoring. *IEEE Inst and Meas*, 12(6):8–15, 2009.

[9] W. Colitti, K. Steenhaut, N. Descouvemont, and A. Dunkels. Satellite based wireless sensor *networks*: Global scale sensing with nano- and pico-satellites. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 445–446, 2008.

[10] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22:1–29, 1997.

[11] J. Green and J. Laffont. Incentives in public decision making. *Studies in Public Economics*, 1:65–78, 1979.

[12] T. Groves. Incentives in teams. *Econometrica*, 1973.

[13] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proc. of HICSS 2000*.

[14] S. Li, Y. Liu, and X. Li. Capacity of large scale wireless networks under gaussian channel model. In *Proc. of MOBICOM 2008*.

[15] Y. Li, X. Li, and P. Wang. A module harvesting wind and solar energy for wireless sensor node. *Advances in Wireless Sensor Networks, the series Communications in Computer and Information Science*, 334:217–2224, 2012.

[16] K. Martinez, R. Ong, and J.K. Hart. Glacsweb: a sensor network for hostile environments. In *Proc. of SECON 2004*.

[17] N. Nisan. Algorithms for selfish agents: Mechanism design for distributed computation. *STACS 1999. LNCS, Meinel, C., Tison, S. (eds.)*, 1563.

[18] N. Nisan and A. Ronen. Algorithmic mechanism design. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing (STOC 1999)*, pages 129–140.

[19] N. Nisan and A. Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35:166–196, 2007.

[20] F. Pavlidou and G. Koltsidas. Game theory for routing modeling in communication networks – a survey. *Journal of Communications and Networks*, 10(3):268–286, Sep. 2008.

[21] R. C. Shah, S. Roy, S. Jain, and W. Brunette. Data mules: Modeling a three-tier architecture for sparse sensor networks. In *Proc. of SNPA 2003*.

[22] Hai-Yan Shi, Wan-Liang Wang, Ngai-Ming Kwok, and Sheng-Yong Chen. Game theory for wireless sensor networks: A survey. *Sensors (Basel, Switzerland)*, 12:9055–97, 12 2012.

[23] A. A. Syed, W. Ye, and J. Heidemann. T-lohi: A new class of mac protocols for underwater acoustic sensor networks. In *Proc. of INFOCOM 2008*. http://www.isi.edu/ilense/snuse/index.html.

[24] B. Tang, N. Jaggi, H. Wu, and R. Kurkal. Energy-efficient data redistribution in sensor networks. *ACM Trans. Sen. Netw.*, 9(2):11:1–11:28, April 2013.

[25] W. Vickrey. Counterspeculation, auctions and competitive sealed tenders. *Journal of Finance*, 1961.

[26] Q. Wangand W. Yang. Energy consumption model for power management in wireless sensor networks. In *Proceedings of SECON 2007*.

[27] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of OSDI 2006*.

[28] Yang Xiao. *Underwater Acoustic Sensor Networks*. Auerbach Publications, 2009.

[29] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *Computer Networks*, 52:2292–2330, 2008.

10. APPENDIX

```
/**
 * Axis
 */

public class Axis {

  private double xAxis;
  private double yAxis;
  private int capa;

  public double getxAxis() {
    return xAxis;
  }
  public void setxAxis(double xAxis) {
    this.xAxis = xAxis;
  }
  public double getyAxis() {
    return yAxis;
  }
  public void setyAxis(double yAxis) {
    this.yAxis = yAxis;
  }
  public int getcapa() {
    return capa;
  }
  public void setcapa(int capa) {
    this.capa = capa;
  }

}


/**
 * Edge
 */
public class Edge{

   private int tail;
   private int head;

   public Edge(int tail, int head, int sort) {
     if (sort == 1) {
        if (tail<=head) {
                this.tail = tail;
                this.head = head;
            } else {
                this.tail = head;
                this.head = tail;
            }
        } else {
            this.tail = tail;
            this.head = head;
        }
    }


    public int getTail(){
        return this.tail;
    }

    public int getHead(){
        return this.head;
    }

    @Override
    public boolean equals(Object o) {
        System.out.println("calling Edge's equals()");
        if(this.tail == ((Edge)o).getTail() && this.head == ((Edge)o).getHead()) {
            return true;
        } else if(this.tail == ((Edge)o).getHead()
                        && this.head == ((Edge)o).getTail()) {
            return true;
        }
        return false;
    }

    @Override
    public String toString(){
        return "(" + tail + ", " + head + ")";
    }
}


/**
 * Link
 */
import java.text.DecimalFormat;
public class Link{
```

```
Edge edge;
double distance;
double Rcost;
double Tcost;
double Scost;
double energy;
DecimalFormat fix = new DecimalFormat("##.######");

public Link(Edge edge, double distance, double Rcost, double Tcost, double Scost
        , double energy) {
   this.edge = edge;
   this.distance = distance;
   this.Rcost = Rcost;
   this.Tcost = Tcost;
   this.Scost = Scost;
   this.energy = energy;
}

public void setEdge(Edge edge) {
   this.edge = edge;
}

public void setDistance(double distance) {
   this.distance = distance;
}

public void setRCost(double Rcost) {
   this.Rcost = Rcost;
}
public void setTCost(double Tcost) {
   this.Tcost = Tcost;
}
public void setSCost(double Scost) {
   this.Scost = Scost;
}

public void setEnergy(double energy) {
   this.energy = energy;
}

public Edge getEdge() {
   return edge;
}
```

```
public double getDistance() {
   return distance;
}

public double getRCost() {
   return Rcost;
}

public double getTCost() {
   return Tcost;
}

public double getSCost() {
   return Scost;
}

public double getEnergy() {
   return energy;
}

@Override
public boolean equals(Object o) {
   if (this.edge.getTail()==((Link)o).getEdge().getTail()
      && this.edge.getHead()==((Link)o).getEdge().getHead()){
      return true;
   }
   return false;
}
@Override
public String toString(){

   return "edge: " + edge.toString() +
         ", distance: " + Math.round(distance * 1000.0)/1000.0 +
         ", receivecost: " + Math.round(Rcost*Math.pow(10,7))/Math.pow(10,7) +
         ", transmitcost: " +Math.round(Tcost*Math.pow(10,7))/Math.pow(10,7) +
         ", storagecost: " + Math.round(Scost*Math.pow(10,7))/Math.pow(10,7) +
         ", energycapacity: " + energy;
}

public int compareTo(Link value) {
   if (this.getEnergy() < value.getEnergy()) {
      return 1;
   } else if (this.getEnergy() > value.getEnergy()) {
      return -1;
   } else {
```

```
            return 0;
        }
    }
}


/**
 * Path
 */
import java.util.ArrayList;

public class Path {
    private ArrayList<Integer> path;
    private double cost;

    public Path(ArrayList<Integer> path, double cost){
        this.path = path;
        this.cost = cost;
    }

    public ArrayList<Integer> getPath() {
        return path;
    }

    public double getCost() {
        return cost;
    }

    public void setPath(ArrayList<Integer> path) {
        this.path = path;
    }

    public void setCost(double cost) {
        this.cost = cost;
    }


    @Override
    public String toString(){

        return this.path.toString() + " " + cost;
    }
}
```

```
/**
 * Dijkstra
 *
 * Implementation of weighed directed edge.
 *
 * Created by marcinkossakowski on 11/2/14.
 */

public class DijkstraFind {
    private int size;
    private HashMap<Integer, Double> weight; // store weights for each vertex
    private HashMap<Integer, Integer> previousNode; // store previous vertex
    private PriorityQueue<Integer> pq; // store vertices that need to be visited
    private WeighedDigraph graph; // graph object

    /**
     * Instantiate algorithm providing graph
     * @param graph WeighedDigraph graph
     */
    public DijkstraFind(WeighedDigraph graph) {
        this.graph = graph;
        size = graph.size();
        //System.out.print(graph.vertices());
    }

    /**
     * Calculate shortest path from A to B
     * @param vertexA source vertex
     * @param vertexB destination vertex
     * @return list of vertices composing shortest path between A and B
     */
    public ArrayList<Integer> shortestPath(int vertexA, int vertexB, int numberOfDG){
        previousNode = new HashMap<Integer, Integer>();
        weight = new HashMap<Integer, Double>();
        pq = new PriorityQueue<Integer>(size, PQComparator);

        /* Set all distances to Infinity */
        for (int vertex : graph.vertices())
            weight.put(vertex, Double.POSITIVE_INFINITY);

        previousNode.put(vertexA, -1); // negative means no previous vertex
        weight.put(vertexA, 0.0); // weight to has to be 0
        pq.add(vertexA); // enqueue first vertex

        while (pq.size() > 0) {
```

```
    int currentNode = pq.poll(); //get the head
    ArrayList<WeighedDigraphEdge> neighbors = graph.edgesOf(currentNode);
            //get the adjacent list of the head

    if (neighbors == null) {
      continue;
    }

    for (WeighedDigraphEdge neighbor : neighbors) {
       int nextVertex = neighbor.to();
       //this loop considers DG can not pass data between themselves
       //if (nextVertex > numberOfDG) {
       double newDistance = weight.get(currentNode) + neighbor.weight();
       if (weight.get(nextVertex) == Double.POSITIVE_INFINITY) {
          previousNode.put(nextVertex, currentNode);
          weight.put(nextVertex, newDistance);
          pq.add(nextVertex);
       } else {
          if (weight.get(nextVertex) > newDistance) {
             previousNode.put(nextVertex, currentNode);
             weight.put(nextVertex, newDistance);
          }
       }
    }
}
/* Path from A to B will be stored here */
ArrayList<Integer> nodePath = new ArrayList<Integer>();

/*
We are reverse walking points to get to the beginning of the path.
Using temporary stack to reverse the order of node keys stored in nodePath.
*/
Stack<Integer> nodePathTemp = new Stack<Integer>();
nodePathTemp.push(vertexB);

int v = vertexB;
while ((previousNode.containsKey(v)) && (previousNode.get(v) >= 0)
                && (v > 0)) {
   v = previousNode.get(v);
   nodePathTemp.push(v);
}

// Put node in ArrayList in reversed order
while (nodePathTemp.size() > 0)
   nodePath.add(nodePathTemp.pop());
```

```
        return nodePath;
    }

    /**
     * Comparator for priority queue
     */
    public Comparator<Integer> PQComparator = new Comparator<Integer>() {

        public int compare(Integer a, Integer b) {
            if (weight.get(a) > weight.get(b)) {
                return 1;
            } else if (weight.get(a) < weight.get(b)) {
                return -1;
            }
            return 0;
        }
    };


}


/**
 * Digraph Edge
 *
 * Implementation of weighed directed edge.
 *
 * Created by marcinkossakowski on 11/2/14.
 */
public class WeighedDigraphEdge {
    private int from, to;
    private double weight;

    /**
     * Construct graph edge
     * @param from
     * @param to
     * @param weight
     */
    public WeighedDigraphEdge(int from, int to, double weight) {
        this.from = from;
        this.to = to;
        this.weight = weight;
    }
    /**
     * @return from vertex
     */
    public int from() {
        return from;
    }

    /**
     * @return to vertex
     */
    public int to() {
        return to;
    }

    /**
     * @return weight of edge between from() and to()
     */
    public double weight() {
        return weight;
    }
}
/**
 * Digraph
 *
 * Implementation of weighed directed edge.
 *
 * Created by marcinkossakowski on 11/2/14.
 */
import java.util.HashMap;
import java.util.ArrayList;
import java.util.HashSet;

// For file reading
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileReader;
import java.util.Map;

public class WeighedDigraph {
    private HashMap<Integer, ArrayList<WeighedDigraphEdge>> adj = new HashMap(); //
adjacency-list

    public WeighedDigraph(Map<String, Link> tree) {
    for (Map.Entry<String, Link> info : tree.entrySet()) {
        addEdge(new WeighedDigraphEdge(info.getValue().getEdge().getHead(),
                info.getValue().getEdge().getTail(),
                info.getValue().getTCost()));
    }
}


    /**
     * Instantiate graph from file with data
     * @param file
     * @throws IOException
     */
    public WeighedDigraph(String file) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(file));
        String line = null;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split("\\s");

            if (parts.length == 3) {
                int from = Integer.parseInt(parts[0]);
                int to = Integer.parseInt(parts[1]);
                double weight = Double.parseDouble(parts[2]);

                addEdge(new WeighedDigraphEdge(from, to, weight));
            }
        }
    }

    /**
     * @param vertex
     * @return list of edges vertex is connected to.
     */
    public ArrayList<WeighedDigraphEdge> edgesOf(int vertex) {
        return adj.get(vertex);
    }

    /**
     * @return list of all edges in the graph.
     */
    public ArrayList<WeighedDigraphEdge> edges() {
        ArrayList list = new ArrayList<WeighedDigraphEdge>();

        for (int from : adj.keySet()) {
            ArrayList<WeighedDigraphEdge> currentEdges = adj.get(from);
            for (WeighedDigraphEdge e : currentEdges) {
```

```
            list.add(e);
        }
    }
    return list;
}

/**
 * @return iterable of all vertices in the graph.
 */
public Iterable<Integer> vertices() {
    HashSet set = new HashSet();
    for (WeighedDigraphEdge edge : edges()) {
        set.add(edge.from());
        set.add(edge.to());
    }

    return set;
}

/**
 * @return size of adjacency list
 */
public int size() { return adj.size(); }

/**
 * @return string representation of digraph
 */
public String toString() {
    String out = "";
    for (int from : adj.keySet()) {
        ArrayList<WeighedDigraphEdge> currentEdges = adj.get(from);
        out += from + " -> ";

        if (currentEdges.size() == 0)
            out += "-,";

        for (WeighedDigraphEdge edge : currentEdges)
            out += edge.to() + " @ " + edge.weight() + ", ";

        out += "\n";
    }

    return out;
}
```

```
    /**
     * Add new edge to the system.
     * @param newEdge
     */
    public void addEdge(WeighedDigraphEdge newEdge) {
        // create empty connection set
        if (!adj.containsKey(newEdge.from()))
            adj.put(newEdge.from(), new ArrayList<WeighedDigraphEdge>());

        ArrayList<WeighedDigraphEdge> currentEdges = adj.get(newEdge.from());

        /* Check if edge already exists,
         * if it is, replace it with new one assuming it needs to be updated */
      //if (!edgeExists)
            currentEdges.add(newEdge);

        adj.put(newEdge.from(), currentEdges);
    }

    /**
     * Graph Tests
     * @param args
     */

}
```

```
/**
 * Sensor Network Graph
 */

import java.awt.*;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Path2D;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Set;

import javax.swing.JFrame;
import javax.swing.JPanel;
public class SensorNetworkGraph extends JPanel implements Runnable {
   private static final long serialVersionUID = 1L;

   private Map<Integer, Axis> nodes;
   private double graphWidth;
   private double graphHeight;
   private int scaling = 25; // 25
   private int ovalSize = 10; // 6
   private int gridCount = 10; // 10
   private boolean connected;
   private Map<Integer, Set<Integer>> adjList;
   private int[] dataGens;

   public SensorNetworkGraph(int[] dataGens){
      this.dataGens = dataGens;
   }

   public boolean isConnected() {
      return connected;
   }

   public void setConnected(boolean connected) {
      this.connected = connected;
   }

   public Map<Integer, Set<Integer>> getAdjList() {
      return adjList;
   }

   public void setAdjList(Map<Integer, Set<Integer>> adjList) {
    this.adjList = adjList;
  }

  public void setNodes(Map<Integer, Axis> nodes) {
    this.nodes = nodes;
    invalidate();
    this.repaint();
  }

  public Map<Integer, Axis> getNodes() {
    return nodes;
  }

  public double getGraphWidth() {
    return graphWidth;
  }

  public void setGraphWidth(double graphWidth) {
    this.graphWidth = graphWidth;
  }

  public double getGraphHeight() {
    return graphHeight;
  }

  public void setGraphHeight(double graphHeight) {
    this.graphHeight = graphHeight;
  }

  @Override
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                   RenderingHints.VALUE_ANTIALIAS_ON);

    double xScale =  ((getWidth() - 3 * scaling) / (graphWidth));
    double yScale =   (( getHeight() - 3 * scaling) / (graphHeight));

    List<Point> graphPoints = new ArrayList<Point>();
    for (Integer key: nodes.keySet()) {
      double x1 = ( nodes.get(key).getxAxis() * (xScale) + (2*scaling));
      double y1 =  ((graphHeight - nodes.get(key).getyAxis())
                                                    * yScale + scaling );
      Point point = new Point();
    point.setLocation(x1, y1);
    graphPoints.add(point);
}

g2.setColor(Color.white);
g2.fillRect(2*scaling, scaling, getWidth() - (3 * scaling)
                                        , getHeight() - 3 * scaling);
g2.setColor(Color.black);


for (int i = 0; i < gridCount + 1; i++) {
    int x0 = 2*scaling;
    int x1 = ovalSize + (2*scaling);
    int y0 = getHeight() - ((i * (getHeight()
                                     - (3*scaling))) / gridCount + (2*scaling));
    int y1 = y0;


    if (nodes.size() > 0) {
        g2.setColor(Color.black);
        g2.drawLine((2*scaling) + 1 + ovalSize, y0
                                                    , getWidth() - scaling, y1);
        String yLabel = ((int) ((getGraphHeight()
                                              * ((i * 1.0) / gridCount))
                                              * 100)) / 100.0 + "";
        FontMetrics metrics = g2.getFontMetrics();
        int labelWidth = metrics.stringWidth(yLabel);
        g2.drawString(yLabel, x0 - labelWidth - 5
                                    , y0 + (metrics.getHeight() / 2) - 3);
    }
    g2.drawLine(x0, y0, x1, y1);
}

for (int i = 0; i < gridCount + 1; i++) {
    int x0 = i * (getWidth() - (scaling * 3)) / gridCount+ (2*scaling);
    int x1 = x0;
    int y0 = getHeight() - (2*scaling);
    int y1 = y0 - ovalSize;
    //if ((i % ((int) ((nodes.size() / 20.0)) + 1)) == 0) {
    if (nodes.size() > 0) {
        g2.setColor(Color.black);
        g2.drawLine(x0, getHeight() - (2*scaling) – 1
                                                        - ovalSize, x1, scaling);
        String xLabel = ((int) ((getGraphWidth() * ((i * 1.0) / gridCount))
                                                * 100)) / 100.0 + "";//i + "";
        FontMetrics metrics = g2.getFontMetrics();
        int labelWidth = metrics.stringWidth(xLabel);
        g2.drawString(xLabel, x0 - labelWidth / 2, y0
                                                + metrics.getHeight() + 3);

    }
    g2.drawLine(x0, y0, x1, y1);
    //}
}

//Draw the edges
Stroke stroke = g2.getStroke();
// pick color
Color darkBlue = new Color(51, 161, 201); // Color white
g2.setColor(darkBlue);
// control the width "?? f"
g2.setStroke(new BasicStroke(1f));
for (int node: adjList.keySet()) {
    if((adjList.get(node) != null) && (!adjList.get(node).isEmpty())) {
        for (int adj: adjList.get(node)) {
            if(adjList.get(node).contains(adj)) {
                int x1 = graphPoints.get(node-1).x;
                int y1 = graphPoints.get(node-1).y;
                int x2 = graphPoints.get(adj-1).x;
                int y2 = graphPoints.get(adj-1).y;
                g2.drawLine(x1, y1, x2, y2);
            }
        }
    }
}

//Draw the oval
g2.setStroke(stroke);
// pick color
Color lightBlue = new Color(61, 89, 171); // Color white
g2.setColor(lightBlue);
for (int i = 0; i < graphPoints.size(); i++) {
    double x = graphPoints.get(i).x - ovalSize / 2;
    double y = graphPoints.get(i).y - ovalSize / 2;
    double ovalW = ovalSize;
    double ovalH = ovalSize;
    Ellipse2D.Double shape = new Ellipse2D.Double(x, y, ovalW, ovalH);

    boolean flag = false;
```

```
            for (int dg: dataGens){
                if(i+1==dg){
                    x = graphPoints.get(i).x;
                    y = graphPoints.get(i).y;
                    g2.fill(createDefaultStar(5, x, y));
                    g2.draw(createDefaultStar(5, x, y));
                    flag = true;
                }
            }

            if (!flag) {
                g2.fill(shape);
                g2.draw(shape);
            }
        }
    }

    public void run() {
        String graphName= "Sensor Network Graph";
        JFrame frame = new JFrame(graphName);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(this);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    private static Shape createDefaultStar(double radius, double centerX, double centerY) {
        return createStar(centerX, centerY, radius, radius * 2.63, 5,
                Math.toRadians(-18));
    }

    private static Shape createStar(double centerX, double centerY,
                                double innerRadius,
                                                double outerRadius, int numRays,
                                double startAngleRad) {
        Path2D path = new Path2D.Double();
        double deltaAngleRad = Math.PI / numRays;
        for (int i = 0; i < numRays * 2; i++) {
            double angleRad = startAngleRad + i * deltaAngleRad;
            double ca = Math.cos(angleRad);
            double sa = Math.sin(angleRad);
            double relX = ca;
            double relY = sa;
            if ((i & 1) == 0)
            {
               relX *= outerRadius;
               relY *= outerRadius;
            }
            else
            {
               relX *= innerRadius;
               relY *= innerRadius;
            }
            if (i == 0)
            {
               path.moveTo(centerX + relX, centerY + relY);
            }
            else
            {
               path.lineTo(centerX + relX, centerY + relY);
            }
         }
         path.closePath();
         return path;
      }
   }
```

```
/**
 * Game Theory
 */

import java.awt.Dimension;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;

import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.xssf.usermodel.XSSFSheet;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import java.text.DecimalFormat;
import ilog.concert.*;
import ilog.cplex.*;
import ilog.cplex.IloCplex.UnknownObjectException;

public class SensorNetworkGameTheroy {

  private static long seed = 995;
  static Random rand = new Random(995);
  static XSSFWorkbook energyworkbook = new XSSFWorkbook();
  static XSSFWorkbook dataworkbook = new XSSFWorkbook();
  static XSSFWorkbook deadnodeworkbook = new XSSFWorkbook();
  static XSSFSheet energysheet = energyworkbook.createSheet("EnergyCostData");
  static XSSFSheet datasheet = dataworkbook.createSheet("DataItems");
  static XSSFSheet deadnodesheet = deadnodeworkbook.createSheet("DeadNode");
  static Map<Integer, Axis> nodes = new LinkedHashMap<Integer, Axis>();
  static Map<Integer, Axis> nodes2 = new LinkedHashMap<Integer, Axis>();
  static Map<Integer, Axis> nodes5 = new LinkedHashMap<Integer, Axis>();
  Map<Integer, Boolean> discovered = new HashMap<Integer, Boolean>();
  Map<Integer, Boolean> explored = new HashMap<Integer, Boolean>();
  Map<Integer, Integer> parent = new HashMap<Integer, Integer>();
  Map<Integer, Integer> connectedNodes = new HashMap<Integer, Integer>();
  Stack<Integer> s = new Stack<Integer>();
  static Map<String, Link> links = new HashMap<String, Link>();
  static Map<String, Link> links2 = new HashMap<String, Link>();
  static Map<String, Link> links3 = new HashMap<String, Link>();
  static Map<String, Link> linkstest = new HashMap<String, Link>();
  static Map<String, Link> linksamp1 = new HashMap<String, Link>();
  static Map<String, Link> linksamp10 = new HashMap<String, Link>();
  static Map<String, Link> linksamp1000 = new HashMap<String, Link>();
  static Map<String, Link> linksamp10000 = new HashMap<String, Link>();
  static Map<String, Link> linksamp1re = new HashMap<String, Link>();
  static Map<String, Link> linksamp10re = new HashMap<String, Link>();
  static Map<String, Link> linksamp1000re = new HashMap<String, Link>();
  static Map<String, Link> linksamp10000re = new HashMap<String, Link>();
  static HashMap<Integer, List<Integer>> close = new HashMap<>();
  static HashMap<Integer, Double> totaldataitems = new HashMap<>();

  static int minCapacity;
  static int capacityRandomRange;
  static int biconnectcounter = 1;
  static int[] dataGens;
  static int[] storageNodes;
  static int[] dataGens2;
  static int[] storageNodes2;
  static int numberOfDG;
  static int numberOfDataItemsPerDG;
  static int numberOfStoragePerSN;
  static int numberOfNodes;
  static DecimalFormat fix = new DecimalFormat("##.########");

  public static void main(String[] args) throws IOException, IloException {

    Scanner scan = new Scanner(System.in);
    System.out.print("The width is set to: ");
    //double width = scan.nextDouble();
    double width = 1000.0;
    System.out.println(width);

    System.out.print("The height is set to: ");
    //double height = scan.nextDouble();
    double height = 1000.0;
    System.out.println(height);

    System.out.print("Number of nodes is set to: ");
    //numberOfNodes = scan.nextInt();
    numberOfNodes = 50;
    System.out.println(numberOfNodes);

    System.out.print("Transmission range in meters is set to: ");
    //int transmissionRange = scan.nextInt();
    int transmissionRange = 250;
    System.out.println(transmissionRange);

    System.out.print("Data Generators amount is set to: ");
    //numberOfDG = scan.nextInt();
    numberOfDG = 10;
    System.out.println(numberOfDG);

    dataGens = new int[numberOfDG];
    System.out.println("Assuming the first " + numberOfDG + " nodes are DGs\n");
    for (int i=1; i<=dataGens.length; i++) {
      dataGens[i-1] = i;
    }

    storageNodes = new int[numberOfNodes-numberOfDG];
    for (int i=0; i<storageNodes.length; i++){
      storageNodes[i] = i + 1 + numberOfDG;
```

```
        }

        System.out.print("Data items per DG is set to: ");
        //numberOfDataItemsPerDG = scan.nextInt();
        numberOfDataItemsPerDG = 100;
        System.out.println(numberOfDataItemsPerDG);

        System.out.print("Data storage per node is set to:");
        numberOfStoragePerSN = scan.nextInt();
        // CHANGE
//      numberOfStoragePerSN = 26;
//      System.out.println(numberOfStoragePerSN);

        capacityRandomRange= 0;

        int numberOfSupDem = numberOfDataItemsPerDG * numberOfDG;
        int numberOfstorage = numberOfStoragePerSN * (numberOfNodes-numberOfDG);
        System.out.println("The total number of data items overloading: " +
                                                                  numberOfSupDem);
        System.out.println("The total number of data items storage: " +
                                                                  ssnumberOfstorage);

        if (numberOfSupDem > numberOfstorage) {
          System.out.println("No enough storage");
          return;
        }

        int numberOfStorageNodes = numberOfNodes - numberOfDG;
        int totalNumberOfData = numberOfDG * numberOfDataItemsPerDG;

        SensorNetworkGameTheroy sensor = new SensorNetworkGameTheroy();
        //sensor.populateNodes(numberOfNodes, width, height);

        File myfile = new File("inputdata.txt");
        readfileNodes(myfile);

        System.out.println("\nNode List:");
        for(int key :sensor.nodes.keySet()) {
          Axis ax = sensor.nodes.get(key);
          System.out.println("Node:" + key + ", xAxis:" + ax.getxAxis() + ", yAxis:" +
                                  ax.getyAxis() + ", energycapacity:" + ax.getcapa());
        }

        Map<Integer, Set<Integer>> adjacencyList1 =
                                                        new LinkedHashMap<Integer, Set<Integer>> ();

sensor.populateAdjacencyList(numberOfNodes, transmissionRange, adjacencyList1);
System.out.println("\nAdjacency List: ");

for(int i: adjacencyList1.keySet()) {
  System.out.print(i);
  System.out.print(": {");
  int adjSize = adjacencyList1.get(i).size();

  if(!adjacencyList1.isEmpty()){
    int adjCount = 0;
    for(int j: adjacencyList1.get(i)) {
      adjCount+=1;
      if(adjCount==adjSize){
        System.out.print(j);
      } else {
        System.out.print(j + ", ");
      }
    }
  }
  System.out.println("}");
}

System.out.println("\nOriginal Graph:");
sensor.executeDepthFirstSearchAlg(width, height, adjacencyList1);
System.out.println();

//test if the graphic is bi-connect
for (int i = 1; i <= numberOfNodes; i++) {
  for (Map.Entry<Integer, Axis> entry : nodes.entrySet()) {
    int k = entry.getKey();
    Axis v = entry.getValue();
    nodes2.put(k, v);
  }
  nodes2.remove(i);
  Map<Integer, Set<Integer>> adjacencyList2 = new LinkedHashMap<Integer,
                                                                      Set<Integer>> ();
  sensor.checkbiconnect(i ,numberOfNodes, transmissionRange, adjacencyList2);
  sensor.executeDepthFirstSearchAlgbi(width, height, adjacencyList2);
}

if(biconnectcounter == 1) {
  System.out.println("\nAll of the Graph is fully connected!");
} else {
  System.out.println("\nSome Graph is not fully connected!!");
  return;
}

//sorting
Map<String, Link> treeMap = new TreeMap<String, Link>(linkstest);

StringBuilder totalenergycost = new StringBuilder();
totalenergycost.append("Sensor Network Edges with Distance, Cost and Capacity:\n");
System.out.println("\nSensor Network Edges with Distance, Cost and Capacity:");
for (Link link : treeMap.values()){
  for (Link innerlink : treeMap.values()) {
    if ((innerlink.getEdge().getHead() == link.getEdge().getHead())
                &&(innerlink.getEdge().getTail() == link.getEdge().getTail())) {
      System.out.println(innerlink.toString());
      totalenergycost.append(innerlink.toString() + "\n");
    }
  }
}

System.out.println();

// run Algorithm
// initiate Lp solver
   // for max function to find the min off load electricity
IloCplex CpFirst = new IloCplex();
IloCplex CpObject = new IloCplex(); // for calculating optimized path

Map<String, IloNumVar> Firstname = varName(CpFirst, treeMap,
                                                   dataGens, storageNodes);
Map<String, IloNumVar> Objtname = varName(CpObject, treeMap,
                                                   dataGens, storageNodes);

System.out.println("Original Objective:");
// create first row
int rowcounter = 0;
Row energyrow = energysheet.createRow(rowcounter);
Row datarow = datasheet.createRow(rowcounter);
Row deadnoderow = deadnodesheet.createRow(rowcounter++);
// name for cols
String[] row = new String[]{"TargetNode", "C_{V-{i}}", "C_V", "dataitems",
                            "dead(Cvi)", "dead(Cv)", "deadNodeLabel",
                            "C_V_fake", "Utility_i_Truth_Telling", "C*_i",
                            "C_i", "Utility_i_Lying", "Die_if_lie",
                            "True_Data_Receive", "True_Data_Send",
                                        "True_Data_Save", "Fake_Data_Receive",
                                        "Fake_Data_Send", "Fake_Data_Save",
                                        "Utility_Difference", "Discarded_amount",
                                        "True_energy_cost", "Single_node_cost"};
// write
for (int i = 0; i < row.length; i++) {
  Cell energycell = energyrow.createCell(i);
  energycell.setCellValue((String) row[i]);
}

energyrow = energysheet.createRow(rowcounter++);
double[] notremove = new double[row.length];
double[] trueDataSend = new double[3];
double[] fakeDataSend = new double[3];
ArrayList<Integer> deadNodes = new ArrayList<>();
ArrayList<Double> totalenergy = new ArrayList<>();
ArrayList<Double> dumtotalenergy = new ArrayList<>();
ArrayList<Double> newCalenergy = new ArrayList<>();
claculateLp(treeMap, adjacencyList1, close, Firstname, Objtname, CpFirst,
                    CpObject, storageNodes, notremove, deadNodes, totalenergy,
                    dumtotalenergy, newCalenergy, trueDataSend,
                    fakeDataSend,"original", "original");
// add the first row (not removing node)

Cell cell = energyrow.createCell(0);
cell.setCellValue((Double) notremove[0]);
Cell cell1 = energyrow.createCell(1);
cell1.setCellValue((Double) notremove[1]);
Cell cell2 = energyrow.createCell(2);
cell2.setCellValue((Double) notremove[1]);
Cell cell3 = energyrow.createCell(3);
cell3.setCellValue((Double) notremove[3]);
Cell cell4 = energyrow.createCell(4);
cell4.setCellValue((Double) notremove[4]);
Cell cell5 = energyrow.createCell(5);
cell5.setCellValue((Double) notremove[4]);

Cell cell6 = energyrow.createCell(6);
cell6.setCellValue(deadNodes.toString());

Cell cell7 = energyrow.createCell(7);
cell7.setCellValue((Double) notremove[1]);

// for remove nodes also generates .xlxs
removeLp(treeMap, adjacencyList1, rowcounter);
```

```
    // write to csv file
    FileOutputStream out = new FileOutputStream(new File("data.xlsx"));
    energyworkbook.write(out);
    out.close();

    System.out.println();
    System.out.println("Finish");
  }

  /**
   * this function create the name for lp solver's col
   * @param treeMap: get the links
   * @param dgs: data generators
   * @param sns: storage nodes
   * @return
   * @throws IloException
   */
  public static Map<String, IloNumVar> varName (IloCplex Cp,
                                                          Map<String, Link> treeMap,
                                                          int[] dgs, int[] sns)
                                                          throws IloException {
    //generateFiles(treeMap, adjacencyList1);
    Map<String, IloNumVar> name = new TreeMap<>();

    // create column names for each index
    for (int i = 1; i <= dgs.length; i++) {
      String str = "x0" + i + "''";
      IloNumVar element = Cp.numVar(0, Double.MAX_VALUE, str);
      name.put(str, element);
    }
    for (Link link : treeMap.values()){
      String str = "x" + link.getEdge().getTail() + "''" +
                                   link.getEdge().getHead() + "'";
      IloNumVar element = Cp.numVar(0, Double.MAX_VALUE, str);
      name.put(str, element);
    }
    for (int i = 0; i < sns.length; i++) {
      String str = "x"+sns[i] + "''" +(dataGens.length + storageNodes.length + 1);
      IloNumVar element = Cp.numVar(0, Double.MAX_VALUE, str);
      name.put(str, element);
    }
    return name;
  }
  public static void removeLp(Map<String, Link> treeMap, Map<Integer, Set<Integer>>
adjacencyList1, int rowcounter) throws IOException, IloException {
    /**
     * fix remove first to get the dataitem to offload
     *
     *
     *
     */
    Scanner scan = new Scanner(System.in);
    System.out.println("Select which amplifier to modify: (1: transfer, 2: receive, 3: save)");
    int method = scan.nextInt();
    System.out.println("Please enter the amplifier:");
    double amp = scan.nextDouble();

    for (int i = dataGens.length+1;i<=(dataGens.length+storageNodes.length); i++) {
      Map<String, Link> temptreeMap = new TreeMap<>(treeMap);
      Map<String, Link> newtreeMap = new TreeMap<>();
      Map<Integer, Set<Integer>> tempadj = new LinkedHashMap<>();
      HashMap<Integer, List<Integer>> tempclose = new HashMap<>();
      double[] tempvalues = new double[8];
      double[] values = new double[8];

      values[0] = i;
      Row energyrow = energysheet.createRow(rowcounter);

      Cell node = energyrow.createCell(0);
      node.setCellValue((int) values[0]);
      for(int k: adjacencyList1.keySet()) {
        if(!adjacencyList1.isEmpty()){
          tempadj.put(k, new HashSet<Integer>());
          for(Integer j : adjacencyList1.get(k)) {
            tempadj.get(k).add(j);
          }
        }
      }

      for(int k: close.keySet()) {
        if(!close.isEmpty()){
          tempclose.put(k, new ArrayList<Integer>());
          for(Integer j : close.get(k)) {
            tempclose.get(k).add(j);
          }
        }
      }
```

```
// change current edge cost (lying) -> only change Tcost
if (method == 1) {
  for (Link link : treeMap.values()) {
    // calculating new cost for amp = 1
    double dis = link.getDistance();
    double newTcost = getTCostOther(dis, amp);
    // only change the cost of i (if edge form )
    // Tail = sender, Head = receiver
    if (link.getEdge().getTail() == i) {
      Link templink = new Link(link.getEdge(), link.getDistance(),
                                          link.getRCost(), newTcost,
                                          link.getSCost(), link.getEnergy());
      newtreeMap.put("(" + link.getEdge().getTail() + ", " +
                                     link.getEdge().getHead() + ")", templink);
    } else {
      Link templink = new Link(link.getEdge(), link.getDistance(),
                                          link.getRCost(), link.getTCost(),
                                          link.getSCost(), link.getEnergy());
      newtreeMap.put("(" + link.getEdge().getTail() + ", " +
                                     link.getEdge().getHead() + ")", templink);
    }
  }
} else if (method == 2){
  for (Link link : treeMap.values()) {
    // calculating new cost for amp = 1
    double dis = link.getDistance();
    double newRcost = getRCostOther(dis, amp);
    // only change the cost of i (if edge form )
    // Tail = sender, Head = receiver
    if (link.getEdge().getHead() == i) {
      // when change receive cost, we needs to change the transfer cost
      double newTcost = link.getTCost() - link.getRCost() + newRcost;
      Link templink = new Link(link.getEdge(), link.getDistance(),
                                          newRcost, newTcost, link.getSCost(),
                                          link.getEnergy());
      newtreeMap.put("(" + link.getEdge().getTail() + ", " +
                                     link.getEdge().getHead() + ")", templink);
    } else {
      Link templink = new Link(link.getEdge(), link.getDistance(),
                                          link.getRCost(), link.getTCost(),
                                          link.getSCost(), link.getEnergy());
      newtreeMap.put("(" + link.getEdge().getTail() + ", " +
                                     link.getEdge().getHead() + ")", templink);
    }
  }
```

```
} else if (method == 3) {
  for (Link link : treeMap.values()) {
    // calculating new cost for amp = 1
    double dis = link.getDistance();
    double newScost = getSCostOther(dis, amp);
    // only change the cost of i (if edge form )
    // Tail = sender, Head = receiver
    if (link.getEdge().getHead() == i) {
      Link templink = new Link(link.getEdge(), link.getDistance(),
                                          link.getRCost(), link.getTCost(),
                                          newScost, link.getEnergy());
      newtreeMap.put("(" + link.getEdge().getTail() + ", " +
                                     link.getEdge().getHead() + ")", templink);
    } else {
      Link templink = new Link(link.getEdge(), link.getDistance(),
                                          link.getRCost(), link.getTCost(),
                                          link.getSCost(), link.getEnergy());
      newtreeMap.put("(" + link.getEdge().getTail() + ", " +
                                     link.getEdge().getHead() + ")", templink);
    }
  }
}

// save the edge cost as a file
StringBuilder totalenergycost = new StringBuilder();
totalenergycost.append("Sensor Network Edges with Distance, Cost and Capacity:\n");
for (Link link : newtreeMap.values()){
  for (Link innerlink : newtreeMap.values()) {
    if ((link.getEdge().getHead() == innerlink.getEdge().getHead())
                  && (link.getEdge().getTail() == innerlink.getEdge().getTail())) {
      totalenergycost.append(innerlink.toString() + "\n");
    }
  }
}
// calculate not yet remove part
ArrayList<Integer> tempdeadNodes = new ArrayList<>();

     // for max function to find the min off load electricity
IloCplex tempCpFirst = new IloCplex();
     // for calculating optimized path
IloCplex tempCpObject = new IloCplex();

// use newtreeMap when remove nodes
Map<String, IloNumVar> tempFirstname = varName(tempCpFirst, newtreeMap,
                                                    dataGens, storageNodes);
Map<String, IloNumVar> tempObjtname = varName(tempCpObject, newtreeMap,
                                                    dataGens, storageNodes);

ArrayList<Double> newstorage = new ArrayList<>();
ArrayList<Double> originalstorage = new ArrayList<>();
ArrayList<Double> newCalenergy = new ArrayList<>();

double[] trueDataSend = new double[3];
double[] fakeDataSend = new double[3];
// fake
claculateLp(newtreeMap, adjacencyList1, close, tempFirstname,
                    tempObjtname, tempCpFirst, tempCpObject, storageNodes,
                    tempvalues, tempdeadNodes, newstorage, originalstorage,
                    newCalenergy, trueDataSend, fakeDataSend,"c" +
                    String.valueOf(i), "c" + String.valueOf(i));

// fake obj
Cell fakeObjvalue = energyrow.createCell(7);
fakeObjvalue.setCellValue((double) tempvalues[1]);

// total cost
double trueEnergyPath = 0;
for (double D : originalstorage) {
  trueEnergyPath += D;
}
Cell trueEfakeObjvalue = energyrow.createCell(21);
trueEfakeObjvalue.setCellValue(trueEnergyPath);

// fake energy cost
Cell fakecost = energyrow.createCell(9);
fakecost.setCellValue((double) newstorage.get(i - 1));

Cell deadNodeList = energyrow.createCell(6);
deadNodeList.setCellValue(tempdeadNodes.toString());

// new dead nodes
Cell deadnodes = energyrow.createCell(4);
deadnodes.setCellValue(tempdeadNodes.size());

// true energy cost (fake path)
Cell truecost = energyrow.createCell(22);
truecost.setCellValue((double) newCalenergy.get(i - 1));

Cell originalEnergy = energyrow.createCell(10);
```

```
        originalEnergy.setCellValue((double) originalstorage.get(i - 1));

        Cell trueDataIn = energyrow.createCell(13);
        trueDataIn.setCellValue((double) trueDataSend[0]);
        Cell trueDataOut = energyrow.createCell(14);
        trueDataOut.setCellValue((double) trueDataSend[1]);
        Cell trueDataSave = energyrow.createCell(15);
        trueDataSave.setCellValue((double) trueDataSend[2]);

        Cell fakeDataIn = energyrow.createCell(16);
        fakeDataIn.setCellValue((double) fakeDataSend[0]);
        Cell fakeDataOut = energyrow.createCell(17);
        fakeDataOut.setCellValue((double) fakeDataSend[1]);
        Cell fakeDataSave = energyrow.createCell(18);
        fakeDataSave.setCellValue((double) fakeDataSend[2]);

        // check die
        // current energy + energy cost of saving one data item > minCapacity , or
        // current energy + energy cost to rely (transfer + receive) data to closest node  >  the
minCapacity
        if (originalstorage.get(i - 1) + treeMap.get("(" + close.get(i).get(0) +
                          ", " + i + ")").getSCost() > minCapacity ||
                  originalstorage.get(i - 1) + treeMap.get("(" + i + ", " +
                          close.get(i).get(0) + ")").getTCost() +
                          treeMap.get("(" + i + ", " + close.get(i).get(0) +
                          ")").getRCost() > minCapacity) {

          Cell deadornot = energyrow.createCell(12);
          deadornot.setCellValue("Dead");

        } else {
          Cell deadornot = energyrow.createCell(12);
          deadornot.setCellValue("OK");
        }

        System.out.println("Removing " + i + " :");
        // remove treeMap which contains the target node
        for (Link link : treeMap.values()){
          if (link.getEdge().getTail() == i) {
            temptreeMap.remove("(" + link.getEdge().getHead() + ", " +
                                link.getEdge().getTail() + ")");
          } else if (link.getEdge().getHead() == i) {
            temptreeMap.remove("(" + link.getEdge().getHead() + ", " +
                                link.getEdge().getTail() + ")");
          }
```

```
}

// remove adjacent list which contains the target node
Set<Integer> set = tempadj.remove(i);
for (Integer target : set) {
  Integer obj = i;
  // test the removing lists
  // System.out.println(target + " " + obj);
  tempadj.get(target).remove(obj);
}

// removing node in closest list
tempclose.remove(i);
for (Integer j : tempclose.keySet()) {
  // if node j's closest node in tempclose is i (the removed node) change
             it
  if (tempclose.get(j).get(0) == i) {
             // find a new closest node, so change the distance to max
    tempclose.get(j).set(1, Integer.MAX_VALUE);
             // get the new adjacent list form tempadj
    Set<Integer> tempset = tempadj.get(j);
    for (int k : tempset) {
      if (temptreeMap.get("(" + j + ", " + k + ")").distance <
                                                   tempclose.get(j).get(1)) {
        tempclose.get(j).set(0, k); // change node
        tempclose.get(j).set(1,
                    (int) temptreeMap.get("(" + j +
                     ", " + k + ")").distance); //change distance
      }
    }
  }
}

// DG will not change
int[] tempDG = dataGens;
// storage will be take out 1 every time
int[] tempSN = new int[storageNodes.length - 1];
int incounter = 0;
int outcounter = 0;

while (incounter < storageNodes.length) {
  if (storageNodes[incounter] == i) {
    incounter++;
  } else {
    tempSN[outcounter++] = storageNodes[incounter++];
    }
  }

// use to test the current storage Nodes

        // for max function to find the min off load electricity
IloCplex CpFirst = new IloCplex();
IloCplex CpObject = new IloCplex(); // for calculating optimized path

// use newtreeMap when remove nodes
Map<String, IloNumVar> Firstname = varName(CpFirst, temptreeMap,
                                           tempDG, tempSN);
Map<String, IloNumVar> Objtname = varName(CpObject, temptreeMap,
                                           tempDG, tempSN);

ArrayList<Integer> deadNodes = new ArrayList<>();
ArrayList<Double> newtotalenergy = new ArrayList<>();
ArrayList<Double> dumtotalenergy = new ArrayList<>();
ArrayList<Double> dumnewCalenergy = new ArrayList<>();
// remove
boolean flag = claculateLp(temptreeMap, tempadj, tempclose, Firstname,
                           Objtname, CpFirst, CpObject, tempSN, values,
                           deadNodes, newtotalenergy, dumtotalenergy,
                           dumnewCalenergy, trueDataSend, fakeDataSend,
                           "r" + String.valueOf(i),
                           "r" + String.valueOf(i));

// new Obj
Cell Objvalue = energyrow.createCell(1);
Objvalue.setCellValue((double) values[1]);

// old Obj is the same as non removed result
double original = energysheet.getRow(1).getCell(2).getNumericCellValue();
Cell Originalvalue = energyrow.createCell(2);
Originalvalue.setCellValue(original);

//new data items
Cell dataitems = energyrow.createCell(3);
dataitems.setCellValue(values[3]);

// old dead node is same as non removed result
double originaldead = energysheet.getRow(1)
                             .getCell(4).getNumericCellValue();
Cell originaldeadnodes = energyrow.createCell(5);
originaldeadnodes.setCellValue(originaldead);
```

```
Cell utiliyTure = energyrow.createCell(8);
double util = (double) values[1] - original;
if (util < 0.01) {
  utiliyTure.setCellValue((double) 0);
} else {
  utiliyTure.setCellValue((double) util);
}

// calculate the fake utility
double fakeUtility = energysheet.getRow(rowcounter)
                                        .getCell(1).getNumericCellValue() –
                                        energysheet.getRow(rowcounter)
                                        .getCell(7).getNumericCellValue() +
                                        energysheet.getRow(rowcounter)
                                        .getCell(9).getNumericCellValue() –
                                        energysheet.getRow(rowcounter)
                                        .getCell(10).getNumericCellValue();

Cell fakeUcell = energyrow.createCell(11);
fakeUcell.setCellValue((double) fakeUtility);

double diff = energysheet.getRow(rowcounter)
                                        .getCell(11).getNumericCellValue() –
                                        energysheet.getRow(rowcounter)
                                        .getCell(8).getNumericCellValue();
Cell originaldata = energyrow.createCell(19);
originaldata.setCellValue((double) diff);

double discardData = 0;
if (energysheet.getRow(rowcounter).getCell(10).getNumericCellValue() >
                                                            minCapacity) {
  double trueData = energysheet.getRow(rowcounter)
                                        .getCell(14).getNumericCellValue() +
                                        energysheet.getRow(rowcounter)
                                        .getCell(15).getNumericCellValue();
  double fakeData = energysheet.getRow(rowcounter)
                                        .getCell(17).getNumericCellValue() +
                                        energysheet.getRow(rowcounter)
                                        .getCell(18).getNumericCellValue();
  discardData = fakeData - trueData;
}

Cell discard = energyrow.createCell(20);
discard.setCellValue((double) discardData);
```

```
      // after writing one row, row++;
      rowcounter++;

      if(!flag) {
        System.out.println("having some problem!");
        return;
      }

      newtreeMap.clear();
      temptreeMap.clear();
      tempadj.clear();
      tempclose.clear();
    }

  }

  static boolean claculateLp(Map<String, Link> treeMap, Map<Integer,
                                    Set<Integer>> adjacencyList1, HashMap<Integer,
                                    List<Integer>> tempclose,
                                    Map<String, IloNumVar> nameFirst,
                                    Map<String, IloNumVar> nameObj, IloCplex CpFirst,
                                    IloCplex CpObj, int[] storges, double[] exldata,
                                    ArrayList<Integer> deadNodes,
                                    ArrayList<Double> faketotalenergy,
                                    ArrayList<Double> truetotalenergy,
                                    ArrayList<Double> newCalenergy,
                                    double[] trueDataSend, double[] fakeDataSend,
                                    String removed, String Lpfilename) throws IOException,
                                    IloException {

    List<IloRange> constraintsFirst = new ArrayList<IloRange>();
    List<IloRange> constraintsObj = new ArrayList<IloRange>();


    // adding first constrain
    for(int i: adjacencyList1.keySet()) {

      IloLinearNumExpr exprFirst = CpFirst.linearNumExpr();
      IloLinearNumExpr exprObj = CpObj.linearNumExpr();

      // if is generators add the source sink
      if (i <= dataGens.length) {
        String dir = "x0" + i + "";
        // for first
    exprFirst.addTerm(1, nameFirst.get(dir));
    // for obj
    exprObj.addTerm(1, nameObj.get(dir));
  }

  // + part
  for(int j : adjacencyList1.get(i)) {
    String dir = "x" + j + """" + i + """";
    exprFirst.addTerm(1, nameFirst.get(dir));
    // for obj
    exprObj.addTerm(1, nameObj.get(dir));
  }

  // - part
  for(int j : adjacencyList1.get(i)) {
    String dir = "x" + i + """" + j + """";
    exprFirst.addTerm(-1, nameFirst.get(dir));
    // for obj
    exprObj.addTerm(-1, nameObj.get(dir));
  }

  // if is storages add the storage sink
  if (i > dataGens.length) {
    String dir = "x" + i + """51";
    exprFirst.addTerm(-1, nameFirst.get(dir));
    // for obj
    exprObj.addTerm(-1, nameObj.get(dir));
  }

  // add constrain
  constraintsFirst.add(CpFirst.addEq(exprFirst, 0));
  constraintsObj.add(CpObj.addEq(exprObj, 0));
}

// adding second constrain
for(int i: adjacencyList1.keySet()) {
  IloLinearNumExpr exprFirst = CpFirst.linearNumExpr();
  IloLinearNumExpr exprObj = CpObj.linearNumExpr();

  // in part
  for(int j : adjacencyList1.get(i)) {
    String dir = "x" + j + """" + i + """";
    exprFirst.addTerm(treeMap.get("("+ j + ", " +
                                          i +")").getRCost(), nameFirst.get(dir));
    exprObj.addTerm(treeMap.get("("+ j + ", " +
                                                         i +")").getRCost(), nameObj.get(dir));
      }
      // out part
      for(int j : adjacencyList1.get(i)) {
        String dir = "x" + i + """ + j + "";
        // when calculating transfer cost, we need to take out the receive cost
                  (from sender)
        double temp = (double)Math.round((treeMap.get("(" + i + ",
                                              " + j + ")").getTCost() –
                                              treeMap.get("(" + o + ", " + j + ")")
                                                                  .getRCost()) * 10000) / 10000;
        exprFirst.addTerm(temp, nameFirst.get(dir));
        exprObj.addTerm(temp, nameObj.get(dir));
      }

      // if is storages add the storage sink
      if (i > dataGens.length) {
        String dir = "x" + i + ""51";
        exprFirst.addTerm(treeMap.get("("+ adjacencyList1.get(i).iterator().next() + ", " + i +
")").getSCost(), nameFirst.get(dir));
        exprObj.addTerm(treeMap.get("("+ adjacencyList1.get(i).iterator().next() + ", " + i +
")").getSCost(), nameObj.get(dir));
      }

      // add constrain
      constraintsFirst.add(CpFirst.addLe(exprFirst, minCapacity));
      constraintsObj.add(CpObj.addLe(exprObj, minCapacity));
    }

    // add constrains for single node ** only firstObj
    // objective needs to be separate since the data is possible to be discarded by
           the data generators (after remove nodes)
    for (Integer i : adjacencyList1.keySet()) {
      IloLinearNumExpr exprFirst = CpFirst.linearNumExpr();

      if (i <= dataGens.length) {
        String dir = "x0" + i + "";
        exprFirst.addTerm(1, nameFirst.get(dir));
        constraintsFirst.add(CpFirst.addLe(exprFirst, numberOfDataItemsPerDG));
      } else {
        String dir = "x" + i + ""51";
        exprFirst.addTerm(1, nameFirst.get(dir));
        constraintsFirst.add(CpFirst.addLe(exprFirst, numberOfStoragePerSN));
      }
    }
```

```
        // add first obj objective function
        IloLinearNumExpr objectiveFirst = CpFirst.linearNumExpr();

        for (int i = 0; i < dataGens.length; i++) {
          int temp = i + 1;
          String dir = "x0" + temp + "";
          objectiveFirst.addTerm(1, nameFirst.get(dir));
        }

        CpFirst.addMaximize(objectiveFirst);

        // only see important messages on screen while solving
        // CpFirst.setParam(IloCplex.Param.Simplex.Display, 0);

        //start solving
        // check if the problem is solvable
        if (CpFirst.solve()) {
          // objective value of min, x, and y
//        System.out.println("obj = " + CpFirst.getObjValue());
//        for (Map.Entry<String, IloNumVar> entry : nameFirst.entrySet()) {
//          System.out.println(entry.getKey() + "  = " + CpFirst.getValue(entry.getValue()));
//        }
//        System.out.println(objectiveFirst.toString());
//        for (int i = 0; i < constraintsFirst.size(); i++) {
//          System.out.println(constraintsFirst.get(i).toString());
//        }

        } else {
          System.out.println("Model not solved");
        }

        // initial data items to offload
        int[] dataIn = new int[numberOfDG];

        Arrays.fill(dataIn, 100);

        // objective value
        System.out.println("Objective value: " + CpFirst.getObjValue());
        if (CpFirst.getObjValue() < 999.999) {
          System.out.println("Can not distribute all data!");
          // single generator constrains start at index 98

          for (int i = 1; i <= dataGens.length; i++) {
```

```
        // get generator's value
        if (CpFirst.getValue(nameFirst.get("x0" + i + "")) < 99.999) {

          System.out.println("x0" + i + "" + ": " +
                                            CpFirst.getValue(nameFirst.get("x0" + i + "")));
          System.out.println("Flag!: ");

          dataIn[i - 1] = (int) CpFirst.getValue(nameFirst.get("x0" + i + ""));
          System.out.println("change to"+ dataIn[i - 1]);
        }
      }
    } else {
      System.out.println("Success!");
    }

    System.out.println(Arrays.toString(dataIn));

    /*------------ Obj's signal node constrains-------------------*/
    for (Integer i : adjacencyList1.keySet()) {
      IloLinearNumExpr exprObj = CpObj.linearNumExpr();

      // dataIn[i] may change if FirstObj is not solvable (DG discard data)
      if (i <= dataGens.length) {
        String dir = "x0" + i + "";
        exprObj.addTerm(1, nameObj.get(dir));
        constraintsObj.add(CpObj.addEq(exprObj, dataIn[i - 1]));
      } else {
        String dir = "x" + i + ""51";
        exprObj.addTerm(1, nameObj.get(dir));
        constraintsObj.add(CpObj.addLe(exprObj, numberOfStoragePerSN));
      }
    }

    // add sencond objective function
    IloLinearNumExpr objectiveObj = CpObj.linearNumExpr();

    for (Link link : treeMap.values()){
      String dir = "x" + link.getEdge().getTail() + "" +
                                    link.getEdge().getHead() + "";
      objectiveObj.addTerm(link.getTCost(), nameObj.get(dir));
    }
    for (int i = 0; i < storges.length; i++) {
      String dir = "x" + storges[i] + ""51";
      int n2 = adjacencyList1.get(storges[i]).iterator().next();
      objectiveObj.addTerm(treeMap.get("("+ n2 + ", " +
                                    storges[i] +")").getSCost(), nameObj.get(dir));
    }

    // write objective function
    CpObj.addMinimize(objectiveObj);

    // only see important messages on screen while solving

    // start solving
    // check if the problem is solvable
    if (CpObj.solve()) {
//      // objective value of min, x, and y
//      System.out.println("obj = " + CpObj.getObjValue());
//      for (Map.Entry<String, IloNumVar> entry : nameObj.entrySet()) {
//        System.out.println(entry.getKey() + "  = " + CpObj.getValue(entry.getValue()));
//      }
//
//      System.out.println(objectiveObj.toString());
//      for (int i = 0; i < constraintsObj.size(); i++) {
//        System.out.println(constraintsObj.get(i).toString());
//      }

    }
    else {
      System.out.println("Model not solved");
    }
    System.out.println("Second Objective value: " + CpObj.getObjValue());
    // file for path
    StringBuilder dataTpath = new StringBuilder();

    dataTpath.append("Second Obj Value: "+ CpObj.getObjValue() + "\n");
    //      double[] tempresult1 = lpObj.getPtrVariables();

    for (Map.Entry<String, IloNumVar> entry : nameObj.entrySet()) {
      dataTpath.append(entry.getKey() + " : " + CpObj.getValue(entry.getValue()) + "\n");
    }
    //getMinFile

    ArrayList<Double> tempp = new ArrayList<>();

    Map<String, Link> originaltreeMap = new TreeMap<String, Link>(linkstest);
    // calculate the true cost
    if (removed.charAt(0) == 'c' && Lpfilename.charAt(0) == 'c') {
      tempp = getMinFile(originaltreeMap, tempclose, CpObj, nameObj, storges,       exldata,
trueDataSend, fakeDataSend,
                        deadNodes, "O" + removed, 0);
      truetotalenergy.addAll(tempp);
      tempp.clear();
    }

    tempp = getMinFile(originaltreeMap, tempclose, CpObj, nameObj, storges,
                        exldata, trueDataSend, fakeDataSend, deadNodes, removed, 1);
    newCalenergy.addAll(tempp);
    tempp.clear();

    // calculate the fake cost or remove cost
    tempp = getMinFile(treeMap, tempclose, CpObj, nameObj, storges, exldata,
                            trueDataSend, fakeDataSend, deadNodes, removed, 0);
    faketotalenergy.addAll(tempp);

    exldata[1] = CpObj.getObjValue();
    exldata[3] = (int) CpFirst.getObjValue();
    // clean up memory used
    CpFirst.end();
    CpObj.end();

    return true;
  }

  /**
   * treemap contains the cost of each edge use to calculate total cost
   * @param treeMap : contains the edge cost
   * @throws IOException
   * @throws IloException
   * @throws UnknownObjectException
   */
  static ArrayList<Double> getMinFile(Map<String, Link> treeMap,
                                        HashMap<Integer, List<Integer>> tempclose,
                                        IloCplex CpIn, Map<String, IloNumVar> cpresult,
                                        int[] storages, double[] exldata,
                                        double[] trueDataSend, double[] fakeDataSend,
                                        ArrayList<Integer> deadNodes, String removed,
                                        int method) throws IOException,
                                        UnknownObjectException, IloException {
    //treeMap.get("("+j+", "+i+")").getRCost() <- format to get cost
    /*
     * getRcost() = receive cost
     * getTcost() = transmit cost
     * getScost() = save cost
     */
    // this map contains cost for each node
    ArrayList<Double> back = new ArrayList<>();
    HashMap<Integer,List<Double>> map = new HashMap<>();
    deadNodes.clear();

    // this map contains Scost for each non-generator node (since save cost cannot be retrieve
from itself)
    HashMap<Integer,Double> nodeScost = new HashMap<>();
    for (Map.Entry<Integer, List<Integer>> pair : tempclose.entrySet()) {
      nodeScost.put(pair.getKey(), treeMap.get("(" + pair.getValue().get(0) +
                                                  ", "+pair.getKey()+")").getSCost());
    }

    // output for each node List: receive cost, transmit cost, store cost;
    // get information form the file (file contains: [transfer node, direction node, how many data
items])
    List<List<Double>> res = new ArrayList<>();
        // temp is for numbers at a line from input file
    List<Double> tempres = new ArrayList<>();
    for (Map.Entry<String, IloNumVar> entry : cpresult.entrySet()){
      String curname = entry.getKey();
      // take out the nodes' lable form name
      // for example curname = x12"11', take out node 12 and 11
      for(int j = 0; j < curname.length(); j++) {
        // check when char is digit
        if (Character.isDigit(curname.charAt(j))) {
          if(curname.charAt(j) == '0'){
            tempres.add((double) 0);
          } else {
            int num = curname.charAt(j) - '0';
            while(j + 1 < curname.length() &&
                                          Character.isDigit(curname.charAt(j + 1))) {
              num = num * 10 + curname.charAt(j + 1) - '0';
              j++;
            }
            tempres.add((double) num);
          }
        }
      }
      // add result value to the last element in tempres and add to res
      tempres.add(CpIn.getValue(entry.getValue()));
      res.add(new ArrayList<>(tempres));
      tempres.clear();
    }
```

```
//initial map
for (int i = 1; i <= numberOfNodes; i++) {
  List<Double> initial = new ArrayList<>();
  initial.add(0.0); // 0 Tcost
  initial.add(0.0); // 1 Rcost
  initial.add(0.0); // 2 Scost
  initial.add(0.0); // 3 total
  map.put(i, initial);
}

// use to check if data items transfered exceed energy capacity
double tempCapaRS = 0.0;
double tempCapaSave = 0.0;

double expectRS = 0.0;
double expectSave = 0.0;

double tempenergy = 0.0;
boolean flag = false;
String tempremoved = removed;
int targetnode = tempremoved.equals("original") ? 0 :
                    Integer.parseInt(tempremoved.replaceAll("[^\\d.]", ""));

// calculate target node's energy cost
for (int i = 0; i < res.size(); i++){
  int transNode = (int) Math.floor(res.get(i).get(0));
  int disNode = (int) Math.floor(res.get(i).get(1));
  double items =  res.get(i).get(2);

  if (removed.equals("original")) {
    totaldataitems.put(transNode,
                              totaldataitems.getOrDefault(transNode, 0.0) + items);
  }

  if (transNode == 0) {
    continue;
  }
  // System.out.println(transNode + " " + disNode); debug
  // calculate non storage node
  if (disNode != numberOfNodes + 1) {
    // case transfer / receive data
    // T cost = sender's transmit cost - receiver's receive cost
    double totalTcost = (treeMap.get("(" + transNode + ", "+
                                                    disNode+")").getTCost() –
                                                    treeMap.get("(" + transNode +", "+
                                                                    disNode+")").getRCost()) * items;
    double totalRcost = treeMap.get("(" + transNode +", " +
                                                                    disNode +")").getRCost() * items;
    // record the energy cost
            // tansferNode's Tcost
    map.get(transNode).set(0, map.get(transNode).get(0) + totalTcost);

            // receiveNode's Rcost
    map.get(disNode).set(1, map.get(disNode).get(1) + totalRcost);
    /*-------------------- calculate data items ------------------------*/
    // when in is our target node
    if (transNode == targetnode && targetnode != 0) {
      // cost for receive + out
      double inoutcost = treeMap.get("(" + transNode+", "+
                                                                    disNode+")").getTCost() * items;
      // still have energy
      if (tempenergy + inoutcost < minCapacity) {
        tempenergy += inoutcost;
        tempCapaRS += items;
      } else { // no energy
        double remainenergy = minCapacity - tempenergy;
        double transfercost = treeMap.get("("+ transNode+", "+
                                                                    disNode+")").getTCost();
        double cansend = remainenergy / transfercost;
        tempCapaRS += cansend;
        if (method == 1 && !flag) {
          trueDataSend[0] = tempCapaRS;
          trueDataSend[1] = tempCapaRS;
          flag = true;
        }
        tempenergy = minCapacity;
      }
      expectRS += items;
    }
  } else {
    // case save data
    // record energy
    map.get(transNode)
                    .set(2, nodeScost.getOrDefault(transNode, 0.0) * items);

    /* ----------------- calculate data items ------------------------ */
    // when in is our target node
    if (transNode == targetnode && targetnode != 0) {
      // cost for receive + out
      double tempcost = nodeScost.getOrDefault(transNode, 0.0) * items;
```

```
                // still have energy
                if (tempenergy + tempcost < minCapacity) {
                  tempenergy += tempcost;
                  tempCapaSave += items;
                } else { // no energy
                  double remainenergy = minCapacity - tempenergy;
                  double transfercost = nodeScost.getOrDefault(transNode, 0.0);
                  double cansend = remainenergy / transfercost;
                  tempCapaSave += cansend;
                  if (method == 1 && !flag) {
                    trueDataSend[2] = tempCapaSave;
                    flag = true;
                  }
                  tempenergy = minCapacity;
                }
                expectSave += items;
              }
            }
          }

          // if not reach the max energy capacity, save the data item
          if (method == 1 && !flag) {
            trueDataSend[0] = tempCapaRS;
            trueDataSend[1] = tempCapaRS;
            trueDataSend[2] = tempCapaSave;
          }
          fakeDataSend[0] = expectRS;
          fakeDataSend[1] = expectRS;
          fakeDataSend[2] = expectSave;

          // output file
          StringBuilder energy_mincostoutput = new StringBuilder();
          energy_mincostoutput.append("The order of the cost: Transfer cost, Receive cost, Save
      cost, total cost, node status").append("\r\n");

          //combine DG and storages
          int[] combine = new int[dataGens.length + storages.length];
          for (int i = 0; i < combine.length; i++) {
            if (i < dataGens.length) {
              combine[i] = dataGens[i];
            } else {
              combine[i] = storages[i - dataGens.length];
            }
          }
```

```
    // calculate total cost (0 + 1 + 2)
    int deadcounter = 0;
    for (int i : combine) {
      double totalcost = map.get(i).get(0) +map.get(i).get(1) + map.get(i).get(2);
      map.get(i).set(3, totalcost);
      energy_mincostoutput.append("Node "+ i + ": ["+ map.get(i).get(0) + ", " +
                                                  map.get(i).get(1) + ", " +
                                                  map.get(i).get(2) + ", "
                                                  map.get(i).get(3))
                                                          .append("], closest node: ")
                                                          .append(tempclose.get(i).get(0));
      // add the energy cost result to send back
      if (method == 1) {
        if (map.get(i).get(3) > minCapacity) {
          back.add((double) minCapacity);
        } else {
          back.add(map.get(i).get(3));
        }
      } else {
        back.add(map.get(i).get(3));
      }
      // calculate weather the node is dead
      if (i <= numberOfDG) { // source nodes
        //  current energy   +   energy cost to rely (transfer + receive) data to closest node  >
the minCapacity user identified
        if (Math.round((map.get(i).get(3) +
                    treeMap.get("(" + i + ", " +
                    tempclose.get(i).get(0) + ")").getTCost() +
                    treeMap.get("(" + i + ", " + tempclose.get(i).get(0) +
                    ")").getRCost()) * 100) >= 100 * treeMap.get("(" +
                    tempclose.get(i).get(0) + ", " + i + ")").getEnergy()) {
          energy_mincostoutput.append(", status: DEAD!").append(";\r\n");
          deadcounter++;
          deadNodes.add(i);
        } else {
          energy_mincostoutput.append(", status: Good").append(";\r\n");
        }
      } else { // storage nodes
        // current energy + energy cost of saving one data item > minCapacity , or
        // current energy + energy cost to rely (transfer + receive) data to closest node  >  the
minCapacity
        if (Math.round((map.get(i).get(3) + treeMap.get("(" +
                    tempclose.get(i).get(0) + ", " + i + ")").getSCost()) * 100) >=
                    100 * treeMap.get("(" + tempclose.get(i).get(0) +
                    ", " + i + ")").getEnergy() || Math.round((map.get(i).get(3) +
                    treeMap.get("(" + i + ", " +
                    tempclose.get(i).get(0) + ")").getTCost() +
                    treeMap.get("(" + i + ", " +
                    tempclose.get(i).get(0) + ")").getRCost()) * 100) >=
                    100 * treeMap.get("(" + tempclose.get(i).get(0) +
                    ", " + i + ")").getEnergy()) {
          energy_mincostoutput.append(", status: DEAD!").append(";\r\n");
          deadcounter++;
          deadNodes.add(i);
        } else {
          energy_mincostoutput.append(", status: Good").append(";\r\n");
        }
      }
    }
    exldata[4] = deadcounter;

    totaldataitems.clear();
    return back;
  }

  /* for different cost analysis */
  // receive and save cost
  double getRSCost(double l){
    final int K = 512; // k = 512B (from paper0)
    final double E_elec = 100 * Math.pow(10,-9); // E_elec = 100nJ/bit (from paper1)
    double Erx = 8 * E_elec * K; // Receiving energy consumption assume is same as saving
    //return Math.round(Erx*100)/100.0; // return the sum of sending and receiving energy
    return Erx*1000; // make it milli J now for better number visualization during calculation
  }

  // transfer cost -> ORIGINAL
  static double getTCost(double l) {
    final int K = 512; // k = 512B (from paper0)
    final double E_elec = 100 * Math.pow(10,-9); // E_elec = 100nJ/bit (from paper1)
    final double Epsilon_amp = 100 * Math.pow(10,-12); // Epsilon_amp = 100
pJ/bit/squared(m) (from paper1)
//      double Etx = E_elec * K + Epsilon_amp * K * l * l; // Transfer energy consumption
    double Etx = E_elec * K * 8 + Epsilon_amp * K * 8 * l * l; //
    //return Math.round(Etx*100)/100.0; // return the sum of sending and receiving energy
    return Math.round(Etx*1000*10000)/10000.0; // make it milli J now for better number
visualization during calculation
  }

  static double getTCostOther(double l, double amp){
    final int K = 512; // k = 512B (from paper0)
    final double E_elec = 100 * Math.pow(10,-9); // E_elec = 100nJ/bit (from paper1)
    final double Epsilon_amp = 100 * amp * Math.pow(10,-12); // Epsilon_amp = 100
pJ/bit/squared(m) (from paper1)
    double Etx = E_elec * K * 8 + Epsilon_amp * K * 8 * l * l; //
    //return Math.round(Etx*100)/100.0; // return the sum of sending and receiving energy
    return Math.round(Etx*1000*10000)/10000.0; // make it milli J now for better number
visualization during calculation
  }

  static double getRCostOther(double l, double amp){
    final int K = 512; // k = 512B (from paper0)
    final double E_elec = 100 * amp * Math.pow(10,-9); // E_elec = 100nJ/bit (from paper1)
    double Erx = 8 * E_elec * K; // Receiving energy consumption assume is same as saving
    //return Math.round(Erx*100)/100.0; // return the sum of sending and receiving energy
    return Erx*1000; // make it milli J now for better number visualization during calculation
  }

  static double getSCostOther(double l, double amp){
    final int K = 512; // k = 512B (from paper0)
    final double E_elec = 100 * amp * Math.pow(10,-9); // E_elec = 100nJ/bit (from paper1)
    double Erx = 8 * E_elec * K; // Receiving energy consumption assume is same as saving
    //return Math.round(Erx*100)/100.0; // return the sum of sending and receiving energy
    return Erx*1000; // make it milli J now for better number visualization during calculation
  }

  //for the original graphic
  void executeDepthFirstSearchAlg(double width, double height,
                                  Map<Integer, Set<Integer>> adjList) {
    s.clear();
    explored.clear();
    discovered.clear();
    parent.clear();
    List<Set<Integer>> connectedNodes = new ArrayList<Set<Integer>>();
    for(int node: adjList.keySet()) {
      Set<Integer> connectedNode = new LinkedHashSet<Integer>();
      recursiveDFS(node, connectedNode, adjList);

      if(!connectedNode.isEmpty()) {
        connectedNodes.add(connectedNode);
      }
    }

    if(connectedNodes.size() == 1) {
      //System.out.println("Graph is fully connected with one connected component.");
    } else {
      System.out.println("Graph is not fully connected");
    }


    //Draw first sensor network graph
    SensorNetworkGraph graph = new SensorNetworkGraph(dataGens);
    graph.setGraphWidth(width);
    graph.setGraphHeight(height);
    graph.setNodes(nodes);
    graph.setAdjList(adjList);
    graph.setPreferredSize(new Dimension(960, 800));
    Thread graphThread = new Thread(graph);
    graphThread.start();

  }

  //for the new graphic (delete nodes to test)
  void executeDepthFirstSearchAlgbi(double width, double height, Map<Integer,
Set<Integer>> adjList) {
    //System.out.println("\nExecuting DFS Algorithm");
    //these have to be clear since they already have elements and values after running the
algorithm
    s.clear();
    explored.clear();
    discovered.clear();
    parent.clear();

    //
    List<Set<Integer>> connectedNodes = new ArrayList<Set<Integer>>();
    for(int node: adjList.keySet()) {
      Set<Integer> connectedNode = new LinkedHashSet<Integer>();
      recursiveDFS(node, connectedNode, adjList);

      if(!connectedNode.isEmpty()) {
        connectedNodes.add(connectedNode);
      }
    }

    if(connectedNodes.size() == 1) {
      //ystem.out.println("Graph is fully connected with one connected component.");
    } else {
      biconnectcounter = biconnectcounter + 1;
      System.out.println("Graph is not fully connected");
    }
  }
```

```
void recursiveDFS(int u, Set<Integer> connectedNode,
                        Map<Integer, Set<Integer>> adjList) {

  if(!s.contains(u) && !explored.containsKey(u)) {
    s.add(u);
    discovered.put(u, true);
  }
  while(!s.isEmpty()) {
    if(!explored.containsKey(u)) {
      List<Integer> list = new ArrayList<Integer>(adjList.get(u));
      for(int v: list) {

        if(!discovered.containsKey(v)) {
          s.add(v);
          discovered.put(v, true);

          if(parent.get(v) == null) {
            parent.put(v, u);
          }
          recursiveDFS(v, connectedNode, adjList);
        } else if(list.get(list.size()-1) == v) {
          if( parent.containsKey(u)) {
            explored.put(u, true);
            s.removeElement(u);

            connectedNode.add(u);
            recursiveDFS(parent.get(u), connectedNode, adjList);
          }
        }
      }
      if(!explored.containsKey(u))
        explored.put(u, true);
      s.removeElement(u);
      connectedNode.add(u);
    }
  }
}

void populateNodes(int nodeCount, double width, double height) {
  // if user want to fix the graphic, enter a number in Random()
  Random random = new Random();

  for(int i = 1; i <= nodeCount; i++) {
    Axis axis = new Axis();
```

```
            int scale = (int) Math.pow(10, 1);
            double xAxis =(0 + random.nextDouble() * (width - 0));
            double yAxis = 0 + random.nextDouble() * (height - 0);
            int capa = random.nextInt(10) + 1;

            xAxis = (double)Math.floor(xAxis * scale) / scale;
            yAxis = (double)Math.floor(yAxis * scale) / scale;


            axis.setxAxis(xAxis);
            axis.setyAxis(yAxis);
            axis.setcapa(capa); //each nodes energy capacity

            nodes.put(i, axis);
        }
    }

    static void readfileNodes(File file) throws IOException {
        // if user want to fix the graphic, enter a number in Random()
        Random random = new Random();
        // original 1312
        Scanner scan = new Scanner(System.in);
        System.out.println("Please enter the energy capacity:");
        minCapacity = scan.nextInt(); //max energy

        FileReader fileReader = new FileReader(file);
        BufferedReader bufferedReader = new BufferedReader(fileReader);
        String line;

        while ((line = bufferedReader.readLine()) != null) {
            Axis axis = new Axis();
            String[] words = line.split("  ");
            int scale = (int) Math.pow(10, 1);
            double xAxis = Double.parseDouble(words[1]);
            double yAxis =Double.parseDouble(words[2]);

            axis.setxAxis(xAxis);
            axis.setyAxis(yAxis);
            axis.setcapa(minCapacity); //each nodes energy capacity

            nodes.put(Integer.parseInt(words[0]) + 1, axis);
        }

        fileReader.close();
```

```
}

void populateAdjacencyList(int nodeCount, int tr,
                                    Map<Integer, Set<Integer>> adjList) {
  for(int i = 1; i <= nodeCount; i++) {
    adjList.put(i, new HashSet<Integer>());
  }

  for(int node1: nodes.keySet()) {
    Axis axis1 = nodes.get(node1);
    for(int node2: nodes.keySet()) {
      Axis axis2 = nodes.get(node2);

      if(node1 == node2) {
        continue;
      }
      double xAxis1 = axis1.getxAxis();
      double yAxis1 = axis1.getyAxis();

      double xAxis2 = axis2.getxAxis();
      double yAxis2 = axis2.getyAxis();

      double distance =  Math.sqrt(((xAxis1-xAxis2)*(xAxis1-xAxis2)) +
                                          ((yAxis1-yAxis2)*(yAxis1-yAxis2)));

      double energy = minCapacity;

      if(distance <= tr) {
        linkstest.put(new String("(" + node2 + ", " + node1 + ")"),
                      new Link(new Edge(node2, node1, 0), distance,
                               getRSCost(distance), getTCost(distance),
                               getRSCost(distance), energy));
        if (!close.containsKey(node2)) {
          List<Integer> list = new ArrayList<>();
          list.add(node1);
          list.add((int) distance);
          close.put(node2, list);
        } else {
          if (close.get(node2).get(1) > distance) {
            close.get(node2).set(0, node1);
            close.get(node2).set(1, (int) distance);
          }
        }
        Set<Integer> tempList = adjList.get(node1);
        tempList.add(node2);
```

```
            adjList.put(node1, tempList);

            tempList = adjList.get(node2);
            tempList.add(node1);
            adjList.put(node2, tempList);
            if (node1 > node2){
              links.put(new String("(" + node2 + ", " + node1 + ")"),
                                     new Link(new Edge(node2, node1, 1), distance,
                                               getRSCost(distance), getTCost(distance),
                                               getRSCost(distance), energy));
            } else {
              links.put(new String("(" + node1 + ", " + node2 + ")"),
                                     new Link(new Edge(node1, node2, 1), distance,
                                               getRSCost(distance), getTCost(distance),
                                               getRSCost(distance), energy));
            }
          }
        }
      }
    }

    //similar as populateAdjacencyList but the number of source nodes are different
    void checkbiconnect(int removeconter,int nodeCount, int tr, Map<Integer, Set<Integer>>
  adjList) {
      int j = 1;
      for(int i=1; i < nodeCount; i++) {
        if (j != removeconter) {
          adjList.put(j, new HashSet<Integer>());
          j++;
        } else {
          j++;
          i=i-1;
        }
      }

      for(int node1: nodes2.keySet()) {
        Axis axis1 = nodes2.get(node1);
        for(int node2: nodes2.keySet()) {

          Axis axis2 = nodes2.get(node2);

          if(node1 == node2) {
            continue;
          }
          double xAxis1 = axis1.getxAxis();
        double yAxis1 = axis1.getyAxis();

        double xAxis2 = axis2.getxAxis();
        double yAxis2 = axis2.getyAxis();

        double distance =  Math.sqrt(((xAxis1-xAxis2)*(xAxis1-xAxis2)) +
                        ((yAxis1-yAxis2)*(yAxis1-yAxis2)));

        double energy = minCapacity;

        if(distance <= tr) {
          Set<Integer> tempList = adjList.get(node1);
          tempList.add(node2);
          adjList.put(node1, tempList);

          tempList = adjList.get(node2);
          tempList.add(node1);
          adjList.put(node2, tempList);
          if (node1 > node2){
                        links2.put(new String("(" + node2 + ", " + node1 + ")"),
                                new Link(new Edge(node2, node1, 1), distance,
                                        getRSCost(distance), getTCost(distance),
                                        getRSCost(distance), energy));
                   } else {
                        links2.put(new String("(" + node1 + ", " + node2 + ")"),
                                new Link(new Edge(node1, node2, 1), distance,
                                        getRSCost(distance), getTCost(distance),
                                        getRSCost(distance), energy));
                   }
        }
      }
    }
  }
}
```