DRE^2: ACHIEVING DATA RESILIENCE IN WIRELESS SENSOR NETWORKS: A QUADRATIC PROGRAMMING APPROACH

_____

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

Shanglin Hsu

Fall 2020

PROJECT: DRE^2: ACHIEVING DATA RESILIENCE IN WIRELESS SENSOR NETWORKS: A QUADRATIC PROGRAMMING APPROACH

AUTHOR: SHANGLIN HSU

APPROVED:

_____

Bin Tang, Ph.D

Project Committee Chair

_____

Liudong Zuo, Ph.D

Committee Member

_____

Mohsen Beheshti, Ph.D

Committee Member

ACKNOWLEDGEMENTS

I would first like to thank my project adviser Dr. Bin Tang, who offers amazing ideas about my research direction and gives critical feedback to help me improve the research. I learned not only the method to do research, but also the way to dive into details about research questions and construct meaningful analyses. I would also like to thank Dr. Liudong Zuo and Dr. Mohsen Beheshti, who attend my master's defense and give helpful feedback about my project.

Shanglin Hsu

Fall 2020

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

We focus on sensor networks that are deployed in challenging environments, wherein sensors do not always have connected paths to a base station, and propose a new data resilience problem. We refer to it as DRE^2: *data resiliency in extreme environments*. As there are no connected paths between sensors and the base station, the goal of DRE^2 is to maximize data resilience by preserving the overflow data inside the network for maximum amount of time, considering that sensor nodes have limited storage capacity and unreplenishable battery power. We propose a quadratic programming-based algorithm to solve DRE^2 optimally. As quadratic programming is NP-hard and takes time to find the optimal solution, we design two time efficient heuristics based on different network metrics. We show via extensive experiments that all algorithms can achieve high data resiliences, while a minimum cost flow-based is most energy-efficient. Our algorithms tolerate node failures and network partitions caused by energy depletion of sensor nodes. Finally we study the feasibility of DRE^2, asking under which circumstances that data resilience is achievable. We put forward a maximum flow-based algorithm to solve it. Underlying our algorithms are flow networks that generalize the edge capacity constraint well-accepted in traditional network flow theory.

1. INTRODUCTION

1.1. Background and Motivation

Data resilience refers to the ability of any network to recover quickly and to continue maintaining availability of data despite of disruptions such as equipment failure, power outage, or malicious attack. Due to resource constraints challenges of wireless sensor networks such as unreplenishable battery power and limited storage capacity of sensor nodes [35], link unreliability and scarce bandwidth of wireless medium [41], and the inhospitable and harsh environments in which they are deployed [5, 6], sensor nodes are often prone to failure and vulnerable of data loss. Therefore, how to ensure that collected data reaches the base station reliably has been an active research topic since the inception of sensor network research. This line of research is usually named under the umbrella of data resilience [20], reliable data transmission [27], or data persistence [23]. We use data resilience throughout the paper.

However, all the existing data resilience research in traditional sensor networks assumes that a base station is always available to collect data, and focuses on how to encode and transmit data to the base station reliably. In this paper, we instead study data resilience from a totally different angle – from emerging sensor network applications wherein a base station is not available to collect the data. Such applications include volcano and seismic sensor networks [26], underground sensor networks [31], underwater or ocean sensor networks [9, 28], and volcano eruption monitoring and glacial melting monitoring [11, 32]. These emerging sensor network applications are designed and deployed to address some of the most fundamental problems facing human beings, such as disaster warning, climate change, and renewable energy.

One common characteristic of these sensor networks is that they are all deployed in challenging or extreme 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. Consequently, it is not practical to deploy data-collecting base stations with power outlets in or near such inaccessible sensor fields. Sensory data generated thus has to be stored inside the network for some unpredictable period of time and then being collected by periodic visits of data mules or mobile sinks [7, 30]. Due to the lack of human intervention and the inadequacy of maintenance in the extreme environments, these sensing applications must operate more resiliently than traditional sensor networks.

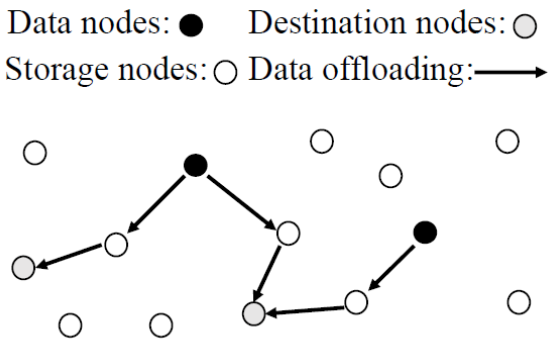


Fig.1. The network model

1.2. Data Resilience Against Sensor Storage Overflow

In this paper, we focus on data resilience against sensor storage overflow, wherein storage spaces of some sensor nodes are depleted therefore they can not store any newly generated data [24, 35]. In our network model, shown in Fig. 1, there are some sensor nodes that are close to the event of interest and generate large amounts of sensory data. As those data cannot be relayed back to base stations in a timely manner due to the extreme environmental conditions, it must be stored locally and thus exhausts the limited storage capacity of these nodes. We refer to the sensor nodes that have exhausted their storage spaces as *data nodes*; the data that cannot be stored locally is

referred to as *overflow data*. Other sensor nodes that have available storage are *storage nodes* (sensor nodes that generate data but still have available storages are considered as storage nodes).

In order to prevent data loss (we assume that any data loss means that the data resilience is not achieved), the overflow data at the data node must be offloaded to the storage nodes to be preserved, and then to be collected when above uploading opportunities become available. The storage nodes that finally store overflow data are *destination nodes*. We refer to this process wherein overflow data is offloaded from data nodes to destination nodes as *data offloading in sensor networks*. As it is not known beforehand when the next uploading opportunity arrives, it is preferred that the offloaded data being stored in destination nodes for longest amount of time before they run out of battery power. Assume that all the sensor nodes have the same energy depleting rates, data thus should be offloaded to destination nodes with high battery power. As each sensor node has limited storage capacity and battery power, the challenge is how to design data offloading scheme that maximizes the survival time for the data packets.

1.3. Contributions

Our contributions are twofold. On the practical side, we identify, formulate, and solve a new algorithmic problem in sensor networks called DRE^2: *data resiliency in extreme environments*. We accurately quantify data resiliences under limited storage capacity and unreplenishable battery power of sensor nodes, and design a quadratic programming (QP)-based algorithm to solve DRE^2 optimally (Section III). As QP is NP-hard and takes time for large scale networks, we design a suite of time-efficient and fault-tolerant heuristic algorithms (Section V). We show that all algorithms achieve high data resiliences while a minimum cost flow (MCF)-based algorithm is most energy efficient (Section VII). Finally we study a relevant and important problem called *feasibility of DRE*^{2} (Section VI). It asks under which conditions that all the overflow data packets

can be offloaded into the network for preservation (i.e., achieving data resilience is feasible). We design a maximum flow (MF)-based algorithm to solve it. Both MCF and MF are formulated as integer linear programs (ILPs).

On the theory side, the underlying enabler of our techniques is flow networks that are delicately converted from the sensor network. These flow networks make possible to identify the convoluted relationship between energy consumption of sensor nodes and the flows of data offloading in DRE^2. As such, we find that in our flow networks, the relationship among flows, capacities, and costs on network edges are significantly different from those well-accepted in conventional network flows. In particular, we generalize the well-known *edge capacity constraint* of flow networks, which mandates that the number of flows on an edge is less than or equal to its capacity. In our designed flow networks, however, we propose that the capacity of an edge must be greater than or equal to the linear combination (i.e., weighted sum) of the flows on this edge. Such *generalized edge capacity constraint* uniquely arises in our flow networks and generalizes aforesaid widely used edge capacity constraint. With this generalization, we are able to apply QPs and ILPs on the flow networks to solve DRE^2 and its related feasibility problem optimally.

2. RELATED WORK

Quadratic programming is the technique of optimizing a quadratic objective function with linear equality and inequality constraints [12, 13], and is one of the simplest forms of non-linear programming. It has been used in sensor network research to solve several important problems such as localization [10, 22], variational data assimilation problem for Lagrangian sensors [10], and optimized transmission for parameter estimation [36]. In contrast, we use this technique to solve a totally different problem. We believe our work is the first one to use quadratic

programming to achieve optimal data resilience in sensor networks deployed in challenging environments.

Data resilience has been an active research since the inception of sensor network research. Ghose et al. [16] achieved resilience by replicating data at strategic locations in the sensor network. Ganesan et al. [14] constructed disjoint multipaths to enable energy efficient recovery from node failures. Recently, network coding techniques are used to recover data from failure-prone sensor networks. Albano et al. [4] proposed in-network erasure coding to improve data resilience to node failures. Kamra et al. [21] proposed to replicate data compactly at neighboring nodes using growth codes that increase in efficiency as data accumulates at the sink. As wireless sensor netowrks utilize sleeping mechanisms to conserve energy, which causes data availability problem, Xu et al. [38] proposed a dataset synchronization protocol in named data networking to achieve data resilience. However, all these research adopts the traditional sensor network model wherein base stations are always available near the networks, therefore are not suitable for the data resilience problem studied in this paper.

Some data resilience research has focused on how to preserve data in disconnection-tolerant sensor network in the absence of base stations. We are aware of two lines of work in this direction. The first line is a sequence of system research [25, 37, 40] that designed cooperative distributed storage systems to improve the utilization of the network's data storage capacity. The other line work is our own research. We took an algorithmic approach and designed a suite of data offloading techniques to achieve different objectives in sensor networks such as minimizing the total energy consumption [5, 35], maximizing the total priorities of preserved data [39], replicating data packets in the events of node failures [5, 34], as well as overcoming the overall storage overflow [33]. However, none of them addressed maximizing data resilience levels and the related

feasibility problem. The closest work to ours is by Hou et al. [19], which achieves data resiliences by maximizing the minimum remaining energy of the destination nodes. Ours is to maximize the sum of the remaining energy of the destination nodes weighted by number of data packets stored on destination nodes, thus is different from their work.

3. PROBLEM FORMULATION OF DRE[2]

3.1. Network Model

We model a sensor network as an undirected graph *G*(*V*,*E*), where $V = \{1, 2, \dots, n\}$ is the set of *n* nodes, and *E* is the set of *m* edges (two nodes are connected if their distance is within the sensor nodes' transmission range). There are l data nodes in the network, denoted as $V_d = \{1, 2, \dots, l\}$. Data node $i \in V_d$ has d_i number of overflow data packets, each of *k* bits. The rest $n - l$ nodes are storage nodes, denoted as $V - V_d = \{l + 1, l + 2, \dots, n\}$. We denote the total $a = \sum_{i=1}^{l} d_i$ overflow data packets as $D = \{D_1, D_2, \dots, D_a\}$. Let the data node of D_j be $dn(j) \in V_d$. Let m_j be the available free storage space at storage node $j \in V - V_d$, meaning that j can further store m_j data packets. We assume naive feasibility condition that $a \leq \sum_{j=l+1}^{n} m_j$ always holds. Otherwise, data loss is inevitable thus data resilience is not achieved.

3.2. Augmented Energy Model

We augment the well-known first order radio model [18] for wireless energy consumption. When node *u* sends a *k*-bit data packet to its one hop neighbor node *v* over their distance $l_{u,v}$ meters, the *transmission energy* spent by u is $E_u^t(v) = \epsilon_{elec} * k + \epsilon_{amp} * k * l_{u,v}^2$, the *receiving energy* spent by v is $E_v^r = \epsilon_{elec} * k$. Here $\epsilon_{elec} = 100nJ/bit$ is the energy consumption per bit on the transmitter circuit and receiver circuit, and $\epsilon_{amp} = 100pJ/bit$ is the energy consumption per bit on the transmit amplifier.

TABLE I. Notation Summary

Notation	Description
V	The set of n data nodes
V_d	$V_d = \{1, 2, \dots, l\}$ is the set of l data nodes, and $V - V_d = \{l+1, l+2, \dots, n\}$ is the set of storage nodes
d_i	Number of overflow data packets from data node $i \in V_d$
m_j	Storage capacity of storage node $j \in V - V_d$
D	$D = \{D_1, D_2, \dots, D_a\}$ is the set of a data packets
$dn(j)$	The data node of $D_j \in D$
E_i	Initial energy level of node i
E_i'	Remaining energy level of node i after data offloading
$E_u^t(v)$	Transmission energy spent by u to transmit a packet to v
E_v^r	Receiving energy spent by v to receive one packet
E_v^s	Storing energy spent by v to store one packet
r	Data offloading function
P_j	The *offloading path* of data packet $D_j \in D$
$\sigma(i,j)$	Node i's successor node in P_j
$y_{i,j}$	Node i's energy cost of offloading data packet D_j
$\xi(i)$	Number of data packets stored at storage node i
$x_{i,j}$	The amount of flows on edge (i,j) in flow networks for QP and ILP
G'	$G'(V', E')$ is the flow network used for QP and MF ILP
G''	$G''(V'', E'')$ is the flow network for MCF ILP

However, this model does not take into account the energy consumption of storing data packets. As write operation costs 13.2μJ amount of energy in Toshiba 128MB flash [3] and we are dealing with large amounts of sensory data, we assume that energy consumption for storing data is non-negligible and augment above first order radio model with storing energy cost. In particular, when storing a data packet, a storage node $v \in V - V_d$ costs $E_v^s = \epsilon_{store} * k$ amount of *storing energy*. Here we assume that $\epsilon_{store} = 100nJ/bit$ is the energy consumption when writing one bit on the memory of a sensor node. Let $E_{u,v} = E_u^t(v) + E_v^r$. We have $E_{v,u} = E_{u,v}$. Note a data node not only transmits all of its own data packets, but also can receive and transmit (i.e., relay) data packets for other data nodes. Meanwhile, a storage node can receive data packets from other nodes and then either transmits or stores them. Table I shows all the notations.

3.3. Problem Formulation

We define *offloading function* as $r : D \to V - V_d$, indicating that data packet $D_j \in D$ is distributed from its data node $dn(j) \in V_d$ to its destination node $r(j) \in V - V_d$. Let $P_j : dn(j), \dots, r(j)$, referred to as the *offloading path* of D_j, be the sequence of distinct sensor nodes along which D_j is offloaded from $dn(j)$ to $r(j)$. Let $\sigma(i,j)$ denote node i's successor node in P_j. Let $y_{i,j}$ be node i's energy cost of offloading data packet D_j, then

$$y_{i,j} = \begin{cases} E_i^t(\sigma(i,j)) & i = dn(j), \\ E_i^r + E_i^s & i = r(j), \\ E_{i,\sigma(i,j)} & i \in P_j - \{dn(j), r(j)\}, \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

When i is the data node of D_j, it costs transmission energy $E_i^t(\sigma(i,j))$; when it is the destination node of D_j, it costs both receiving energy E_i^r and storing energy E_i^s; when it is a relaying node of D_j, it costs both receiving and transmission energy, the sum of which is $E_{i,\sigma(i,j)}$. Otherwise, node i is not involved in D_j's offloading thus costs zero amount of energy. Let E_i and E_i' denote sensor node i's initial energy level and remaining energy after all the a data packets are offloaded, respectively. Then, $E_i' = E_i - \sum_{j=1}^{a} y_{i,j}, \ \forall \ i \in V$.

Definition 1: (Data Resilience Levels (DRLs).) Given a sensor network G with a data packets to be offloaded, its *data resilience level* (DRL), denoted as $D(G)$, is defined as the sum of remaining energy of the destination nodes of all the a data packet, i.e., $D(G) = \sum_{j=1}^{a} E_{r(j)}'$. It is also the case that $D(G) = \sum_{i=l+1}^{n} (E_i' \times \xi(i))$, where $\xi(i)$ is the number of data packets that are finally stored at storage node i.

$D(G)$ indicates the network's best achievable effort to preserve all a data packets, as the more energy of a storage node, the longer time its stored data can survive. The objective of DRE[2] is to find a offloading function r and a set of offloading paths $P = \{P_1, P_2, \dots, P_a\}$ to offload the a

data packets to their destination nodes, such that the DRL of the network is maximized after offloading, i.e., $max_{r,P}\ D(G)$, under the energy constraint of sensor nodes: $E_i' \geq 0, \forall\ i \in V$ and the storage capacity constraint of sensor nodes: $|\{j \mid r(j) = i, 1 \leq j \leq a\}| \leq m_i, \forall\ i \in V - V_d$.

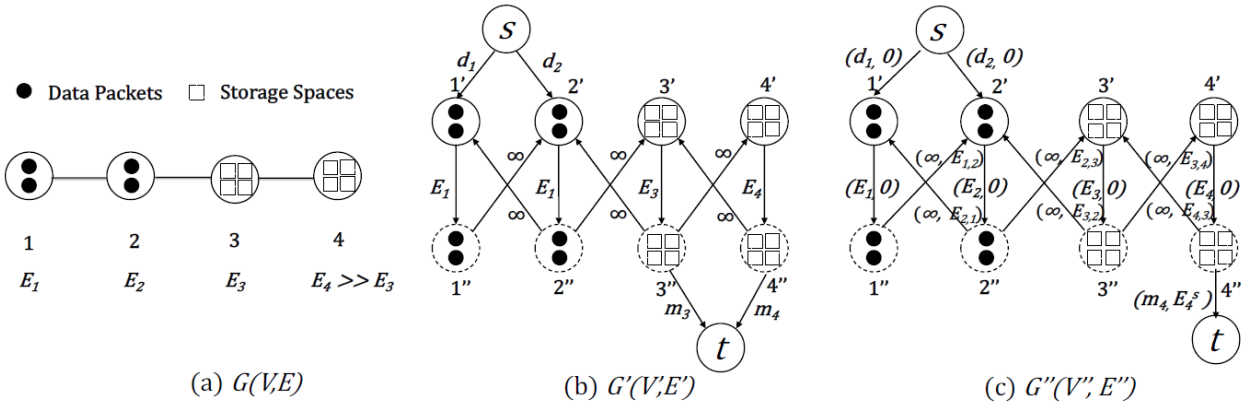


Fig. 2. Network Models: (a) shows a linear sensor network $G(V, E)$ with two data nodes 1 and 2, each having two data packets to offload, and two storage nodes 3 and 4, each having four storage spaces. (b) shows its converted flow network $G'(V', E')$ for QA (A) that maximizes DRLs and ILP (C) that finds maximum number of data packets offloaded. (c) shows its converted flow network $G''(V'', E'')$ for ILP (B) that finds the minimum energy cost of data offloading. As $E_4 > E_3$ and node 4 has enough storage to store all the four data packets, the set of high-energy storage nodes $V_h = \{4\}$.

EXAMPLE 1: Fig. 2(a) shows a linear sensor network with four nodes: 1 and 2 are data nodes, each has two overflow data packets; 3 and 4 are storage nodes, each has storage capacity of four. $E_4 \gg E_3$. To achieve maximum DRL, the optimal solution is to offload all the four data packets to node 4, even though it costs more energy than offloading to node 3. We use this example to illustrate our QP and ILP solutions next.

4. QUADRATIC PROGRAMMING (QP) SOLUTION

To maximize the DRL of any given instance of DRE[2], the fundamental challenge is to find each data packet's destination node as well as the offloading path from its data node to this destination node. Then we are able to compute the number of data packets each destination node stores as well as its remaining energy level, therefore calculating the DRL. In particular, we need to represent energy levels, data packets, and storage capacities in a way that they can be computed-network flow modeling [29] is particularly suitable for such representation, as demonstrated below.

4.1. Graph Conversion

We first convert the sensor network graph $G(V, E)$ in Fig. 2(a) to a flow network $G'(V', E')$ in Fig. 2(b).

I. 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.

II. 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 incoming directed edges of node *i* are incident on i' and all outgoing directed edges of node *i* emanate from i''. Therefore the two directed edges (i, j) and (j, i) in Step I are now changed to (i'', j') and (j'', i').

III. Add a super source node *s*, and connect *s* to the in-node i' of the data node $i \in V_d$ with an edge. Set the capacity of this edge as d_i, the number of data packets at data node *i*.

IV. Add a super sink node *t*, and connect out-node i'' of the storage node $i \in V - V_d$ to *t*. Set its edge capacity m_i, the storage capacity of storage node *i*.

Hence, $V' = \{s\} \cup \{t\} \cup \{i' : i \in V\} \cup \{i'' : i \in V\}$ and $E' = \{(s, i') : i \in V_d\} \cup \{(i', i'') : i \in V\} \cup \{(i'', j') : (i, j) \in E\} \cup \{(j'', i') : (i, j) \in E\} \cup \{(i'', t) : i \in V - V_d\}$. We have $|V'| = 2n + 2$ and $|E'| = 2m + 2n$. Above conversion techniques are used in our previous work [19,39] that solving related data preservation problems in sensor networks.

Rationale of the Conversion. The rationale of above conversion is fourfold. First, as the flows start from *s* and end at *t* in flow network G', and *s* connects to in-nodes of data nodes while *t* connects to out-nodes of storage nodes, it "forces" that overflow data packets are offloaded from data nodes to storage nodes. Second, with the node-splitting and the initial energy levels now being capacities of newly created edges, it guarantees that each node cannot spend more energy in data offloading than it has. Third, with the d_i and m_i now being "encoded" as the capacities of edges connecting *s* and i' (for data nodes) and i'' to *t* (for storage nodes), it makes sure that a data node cannot offload more than it has and a storage node cannot store more than its storage allows. Fourth, and most importantly, the energy consumption of each node can be accurately computed using flows in G', as shown below.

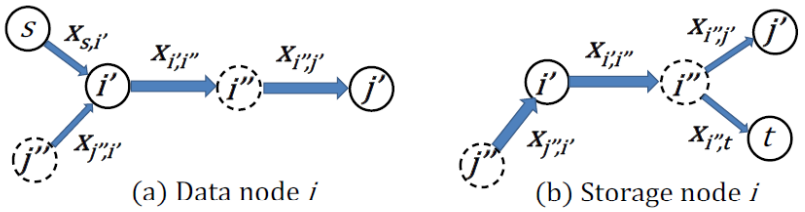


Fig. 3. Node Flows: (a) A data node transmits its own and relays data packets from others and (b) a storage node relays or stores the data packets from others.

4.2. Computing Energy Consumptions of Nodes

Let $x_{i,j}$ be the amount of flows on directed edge (i, j) in G'. Recall that although both data nodes and storage nodes can receive data packets from other sensor nodes, the difference is that of all the received data packets, a data node must transmit all of them whereas a storage node can

store some of them as long as its storage allows. Besides, a data node must transmit all of its own overflow data packets to others. Fig. 3(a) and (b) show the flows in G' that go through a data node and a storage node *i* respectively. We have below observations about the flows and their incurred energy cost, and the flow conservations.

- **Observation 1 (for both data and storage nodes).** Given any node $i \in V$ and any of its neighboring node *j* (i.e., $(i,j) \in E$)[1], the number of data packets *i* receives from *j* is $x_{j'',i'}$, the number of data packets *i* transmits to *j* is $x_{i'',j'}$. Thus, node *i*'s total receiving energy cost is $E_i^r \times \sum_{j:(i,j)\in E} x_{j'',i'}$ and its total transmission energy cost is $\sum_{j:(i,j)\in E}(E_i^t(j) \times x_{i'',j'})$.
- **Observation 2 (for data nodes).** For any data node $i \in V_d$, the number of its own data packets that it transmits is $x_{s,i'}$. As the data packets *i* transmits are either its own or received from others, we have $x_{s,i'} + \sum_{j:(i,j)\in E} x_{j'',i'} = \sum_{j:(i,j)\in E} x_{i'',j'}$.
- **Observation 3 (for storage nodes).** For any storage node $i \in V - V_d$, the number of data packets it stores is $x_{i'',t} = \xi(i)$ (recall $\xi(i)$ is the number of data packets that are finally stored at storage node *i*). As the data packets *i* receives are either transmitted to other nodes or stored at *i*, we have $\sum_{j:(i,j)\in E} x_{j'',i'} = \sum_{j:(i,j)\in E} x_{i'',j'} + x_{i'',t}$. The total storing energy cost of *i* is $E_i^s \times x_{i'',t}$.

[1] Note that it is not $(i,j) \in E'$, as all the data nodes, storage nodes, and neighboring nodes are nodes in G', not G''.

Therefore, the DRL can be represented as below:

$$D(G) = \sum_{i=l+1}^{n} (\xi(i) \times E_i')$$

$$= \sum_{i \in V - V_d} \left(x_{i'',t} \cdot \left(E_i - E_i^r \times \sum_{j:(i,j) \in E} x_{j'',i'} - \sum_{j:(i,j) \in E} \left(E_i^t(j) \times x_{i'',j'} \right) - E_i^s \times x_{i'',t} \right) \right). \quad (2)$$

As D(G) is a concave quadratic expression, DRE^2 can thus be represented as below QP formulation (A):

$$(A) \quad \max \; D(G) \quad (3)$$

s.t

$$x_{s,i'} = d_i, \qquad \forall\, i \in V_d \quad (4)$$

$$x_{i'',t} \le m_i, \qquad \forall\, i \in V - V_d \quad (5)$$

$$x_{s,i'} + \sum_{j:(i,j) \in E} x_{j'',i'} = \sum_{j:(i,j) \in E} x_{i'',j'}, \qquad \forall\, i \in V_d \quad (6)$$

$$\sum_{j:(i,j) \in E} x_{j'',i'} = \sum_{j:(i,j) \in E} x_{i'',j'} + x_{i'',t}, \qquad \forall\, i \in V - V_d \quad (7)$$

$$E_i^r \times \sum_{j:(i,j) \in E} x_{j'',i'} + \sum_{j:(i,j) \in E} (E_i^t(j) \times x_{i'',j'}) \le E_i, \qquad \forall\, i \in V_d \quad (8)$$

$$E_i^r \times \sum_{j:(i,j) \in E} x_{j'',i'} + \sum_{j:(i,j) \in E} \left(E_i^t(j) \times x_{i'',j'} \right) + E_i^s \times x_{i'',t} \le E_i, \qquad \forall\, i \in V - V_d \quad (9)$$

Eqn. 4 mandates that to achieve data resilience, data node *i* must be able to offload all its d_i number of data packets into the network (we will study the feasibility problem of DRE^2 in Sec. VI). Inequality 5 shows the storage constraint of storage nodes. Eqn. 6 and 7 show the flow

conservation for data nodes and storage nodes, respectively (Observations 2 and 3). Inequalities 8 and 9 represent the energy constraint of data nodes and storage nodes respectively.

Generalized Edge Capacity Constraint. Inequalities 8 and 9 need some special notes. In Fig. 2(b) and 2(c), the amount of flows on edge (i', i'') (i.e., $x_{i', i''}$) is not simply less than or equal to the capacity of (i', i'') (i.e., E_i), as stipulated by the edge capacity constraint in conventional network flows. Instead, the relationship between $x_{i', i''}$ and E_i is more intricate, as shown in Inequalities 8 and 9. 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 (l.h.s of Eqn. 6). $x_{i', i''}$ also equals to the total amount of data packets it transmits (r.h.s of Eqn. 6). As such, the energy cost of i can now be represented as a function of $x_{j'', i'}$ and $x_{i'', j'}$ (l.h.s. of Inequality 8). In other words, the total flows on (i', i'') are expressed as the linear combinations (i.e., weighted sum) of its constituent sub-flows. Similar observations can be made for storage node i, except the flow $x_{s, i'}$ in data node is now $x_{i'', t}$ in storage node. We refer to Inequalities 8 and 9 as *generalized edge capacity constraint*, and formally define it as below.

Definition 2: (**Generalized Edge Capacity Constraint.**) In a flow network, given any edge (u, v), let $f(u, v)$ and $cap(u, v)$ denote its flows and capacity respectively. The generalized capacity constraint stipulates that $\sum_{i=1}^{|f(u,v)|} a_i \leq cap(u, v)$ where a_i is the weight for i^{th} flow on the edge.

When a_i= 1 for all the flows, the generalized edge capacity constraint becomes $f(u, v) \leq cap(u, v)$, the traditional edge capacity constraint. By generalizing this widely used constraint in flow network, we believe our work augments the network flow model and can have an impact on its related theory.

Solving QP. QP can be solved by the classic Wolfe's modified simplex method [13], which is based on solving a system of linear relations subject to complementarity conditions. There are many production QP solvers such as CGAL [1] and CPLEX [2]. We adopt CPLEX due to its performances. Besides, CPLEX can improve the efficiency of QP by allowing *gap tolerance* to find a feasible solution quickly (more in Section VII). As QP is NP-hard [15], we design two time-efficient heuristic algorithms below and show via experiments that they perform close to the optimal QP solution.

5. HEURISTIC ALGORITHMS

To maximize DRLs, an intuitive solution is to offload data packets to nodes with initial high energy levels, defined below.

Definition 3: (**High-Energy Storage Nodes.**) High-energy storage nodes, denoted as V_h, are the set of storages nodes with the highest initial energy levels that can store all the a data packets. More formally, we sort storage nodes $V - V_d$ in non-ascending order of their initial energy: $E_{v_1} \geq E_{v_2} \geq \cdots \geq E_{v_{n-1}}$. Then the top k + 1 nodes $\{v_1, \ldots, v_k, v_{k+1}\}$ where $\sum_{i=1}^{k} m_{v_i} < a \leq \sum_{i=1}^{k+1} m_{v_i}$ is V_h.

Both below algorithms are centered around how to offload data packets to V_h in an energy-efficient manner.

5.1. Network-Based Algorithm

For each storage node $i \in V_h$, Algo. 1 finds m_i data packets that are closest to i and offloads them to i via the currently available shortest path (in terms of energy consumption). Its time complexity is $O(n^2)$.

Algorithm 1: Network-Based Algorithm.
Input: A sensor network *G* with m_i, E_i, data packets *D*;
Output: $D(G)$;
1. Compute $V_h = \{v_1, \dots, v_k, v_{k+1}\}$;
2. **for** $(1 \le i \le k)$
3. Find the m_i data packets that are closest to v_i and offload them to v_i via the current shortest path between each data packet and v_i;
4. Update the energy levels of all the nodes on the path;
5. **end for**;
6. Offload each of the $a - \sum_{i=1}^{k} m_{v_i}$ data packets to v_{k+1} via shortest path and update the energy levels;
7. Compute $D(G) = \sum_{i=l+1}^{n} (E_i' \times \xi(i))$;
8. **RETURN** $D(G)$.

5.2. Minimum-Cost-Flow (MCF)-Based Algorithm

Although Algo. 1 saves energy by offloading data packets to their closest nodes in V_h, it does not consider global energy minimization in data offloading. Below we design Algo. 2 and prove that is minimizes total energy consumption in data offloading. It is a MCF-based algorithm applied on another properly converted flow network $G''(V'', E'')$ from the sensor network $G(V, E)$. In MCF, each edge in the flow network has a capacity and a cost and the goal is to minimize the total cost of the flows.

We first present the conversion and then the MCF ILP. As the first two steps of the conversion are the same as the one in Section IV, we start with Step IIII below:

III. For directed edge connecting *s* to the in-node $i'$of the data node $i \in V_s$, set its capacity as d_i and its cost as zero.

IV. For directed edge (i', i''), set its capacity as E_i and cost as zero.

V. For directed edge (i'', j'), set its capacity as infinity and cost as $E_{i,j} = E_i^t(j) + E_j^r$, the sum of node *i*'s transmitting energy and node *j*'s receiving energy. For (j'', i'), set its

capacity as infinity and cost as $E_{j,i} = E_j^t(i) + E_i^r$, the sum of node *j*'s transmitting energy and node *i*'s receiving energy.

VI. For directed edge connecting the out-node i'' of the high-energy storage node $i \in V_h$ to *t*, set its capacity as m_i, the storage capacity of *i*, and its cost as E_i^s, the energy cost of storing one data packet by *i*.

With above transformation, the sensor network $G(V, E)$ in Fig. 2(a) is now converted to a flow network $G''(V'', E'')$ in Fig. 2(c). We next present the ILP formulation (B) for MCF.

$$(B) \quad \min \sum_{(i,j)\in E''} x_{i,j} \times c_{i,j} \tag{10}$$

s.t

$$x_{s,\,i'} = d_i, \qquad \forall\, i \in V_d \tag{11}$$

$$x_{i'',t} \le m_i, \qquad \forall\, i \in V_h \tag{12}$$

$$x_{s,i'} + \sum_{j:(i,j)\in E} x_{j'',\,i'} = \sum_{j:(i,j)\in E} x_{i'',\,j'}, \qquad \forall\, i \in V_d \tag{13}$$

$$\sum_{j:(i,\,j)\in E} x_{j'',\,i'} = \sum_{j:(i,\,j)\in E} x_{i'',\,j'} + x_{i'',t}, \qquad \forall\, i \in V_h \tag{14}$$

$$E_i^r \times \sum_{j:(i,\,j)\in E} x_{j'',\,i'} + \sum_{j:(i,\,j)\in E} (E_i^t(j) \times x_{i'',\,j'}) \le E_i, \quad \forall\, i \in V_d \tag{15}$$

$$E_i^r \times \sum_{j:(i,\,j)\in E} x_{j'',\,i'} + \sum_{j:(i,\,j)\in E} \left(E_i^t(j) \times x_{i'',\,j'}\right) + E_i^s \times x_{i'',\ t} \le E_i, \quad \forall\, i \in V_h \tag{16}$$

$$\sum_{j:(i,\,j)\in E} x_{j'',\,i'} = \sum_{j:(i,\,j)\in E} x_{i'',\,j'}, \qquad \forall\, i \in V - V_d - V_h \tag{17}$$

$$E_i^r \times \sum_{j:(i,\,j)\in E} x_{j'',\,i'} + \sum_{j:(i,\,j)\in E} \left(E_i^t(j) \times x_{i'',\,j'}\right) \le E_i, \quad \forall\, i \in V - V_d - V_h \tag{18}$$

In the objective function 10, $x_{i,j}$ and $c_{i,j}$ are the amount of flows and cost on edge $(i,j) \in E''$, respectively. The Constraints 11-16 are similar to those in quadratic programming (A), except that Constraints 12, 14, and 16 are now applied on V_h, as only high-energy storage nodes V_h can store data packets. Finally, we add two more constraints viz. Equation 17 and Inequality 18 to respectively address the flow conservation and energy constraint of all the storage nodes that are not in V_h. Algo. 2 below calls ILP (B) as a subroutine:

Algorithm 2: MCF-Based Algorithm.
Input: A sensor network *G* with m_i, E_i, and *D*;
Output: $r: D(G)$;
1. Compute $V_h = \{v_1, \dots, v_k, v_{k+1}\}$;
2. Convert $G(V,E)$ to flow network $G''(V'', E'')$;
3. Compute ILP (B) on G'';
4. Compute $D(G) = \sum_{i=l+1}^{n}(E_i' \times \xi(i))$;
5. **RETURN** D(*G*).

Theorem 1: Algo. 2 achieves minimum energy consumption in offloading *a* data packets to nodes in V_h.

Proof: By applying MCF algorithm on G'', it guarantees that all the *a* overflow data packets are offloaded with minimum total energy cost while respecting the storage and energy constraints of sensor nodes.

Solving MCF. We implement MCF ILP using CPLEX [2]. MCF can also be solved efficiently and optimally by combinatorial algorithms such as scaling push-relabel proposed by Goldberg [17]. Its time complexity is $O(a^2 \cdot b \cdot \log(a \cdot c))$, where *a*, *b*, and *c* are number of nodes, number of edges, and maximum edge capacity in the flow network, respectively.

All QP, Network- and MCF-based algorithms are fault-tolerant. Even with node failures and network partitions caused by energy depletion of sensor nodes, they can still achieve high DRLs and energy-efficiency as will be shown in Section VII.

6. FEASIBILITY PROBLEM OF DRE^2

Due to aforesaid node failures and the resulted network partitions between data nodes and storage nodes, offloading all the data packets into the network is not always possible. For example, in Fig. 2(a), if E_3 is small enough, node 3 does not have enough energy to either store packets from nodes 1 and 2 or relay them to node 4, failing to achieve data resilience. We thus tackle an important *feasibility problem*: Given any instance of DRE^2, can all the *a* data packets be offloaded?

We answer this question by finding the maximum number of data packets offloaded, which is solved by below ILP (C) on the flow network $G'(V', E')$ in Fig. 2(b).

$$(A) \quad \max \sum_{i \in V_d} x_{s,i'} \tag{19}$$

s.t

$$x_{s,\,i'} \le d_i, \qquad \forall\, i \in V_d \tag{20}$$

$$x_{i'',t} \le m_i, \qquad \forall\, i \in V - V_d \tag{21}$$

$$x_{s,i'} + \sum_{j:(i,j)\in E} x_{j'',\,i'} = \sum_{j:(i,j)\in E} x_{i'',\,j'}, \qquad \forall\, i \in V_d \tag{22}$$

$$\sum_{j:(i,\,j)\in E} x_{j'',\,i'} = \sum_{j:(i,\,j)\in E} x_{i'',\,j'} + x_{i'',t}, \qquad \forall\, i \in V - V_d \tag{23}$$

$$E_i^r \times \sum_{j:(i,\,j)\in E} x_{j'',\,i'} + \sum_{j:(i,\,j)\in E} (E_i^t(j) \times x_{i'',\,j'}) \le E_i, \quad \forall\, i \in V_d \tag{24}$$

$$E_i^r \times \sum_{j:(i,\,j)\in E} x_{j'',\,i'} + \sum_{j:(i,\,j)\in E} \left(E_i^t(j) \times x_{i'',\,j'}\right) + E_i^s \times x_{i'',\,t} \le E_i, \quad \forall\, i \in V - V_d \tag{25}$$

Objective Function 19 is to find the maximum flow; i.e., the maximum number of data packets that can be offloaded. All the constraints 20-25 are similar to those in QP (A), except that

Eqn. 4 is changed to Inequality 20, as now it is not always possible to offload all the *a* data packets. We give below theorem regarding the feasibility of DRE^2.

Theorem 2: Given any instance of DRE^2 in *G*, if $\sum_{i \in V_d} x_{s,i'} = a$ in G', then achieving data resilience is feasible.

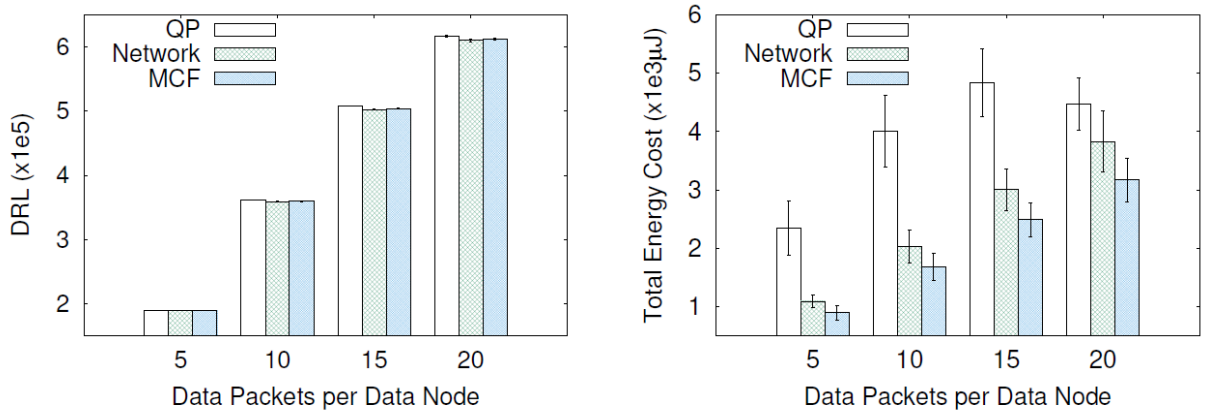
Proof: We have $\sum_{i \in V_d} d_i = a$. As $x_{s,i'} \leq d_i$ in G', we have $\sum_{i \in V_d} x_{s,i'} = \sum_{i \in V_d} d_i \leq a$. When $\sum_{i \in V_d} x_{s,i'} = a$, it must be that $x_{s,i'} = d_i$ for any $i \in V_d$, meaning each data node i successfully offloads its d_i data packets. Therefore, all *a* data nodes are successfully offloaded.

Solving Maximum Flow. We implement maximum flow ILP using CPLEX [2]. Meanwhile, there are also two kinds of well-known combinatorial maximum flow algorithms viz. augmenting path and push-relabel [8]. Both algorithms are strongly polynomial while push-relabel is in general more flexible and efficient than augmenting path [29]. The time complexity of push–relabel maximum flow algorithm is $O(|V'|^2 \cdot |E'|)$for a flow network $G'(V', E')$.

7. PERFORMANCE EVALUATION

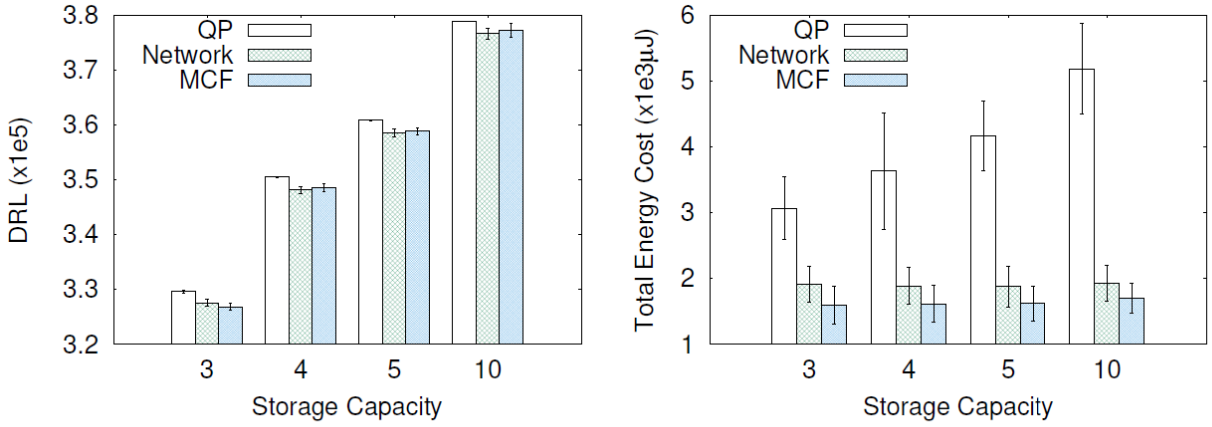
We compare the performance of different algorithms viz. QP-based (referred to as **QP**), Network-Based (referred to as **Network**), and MCF-Based (referred to as **MCF**). We consider both small scale networks of 50 sensor nodes (10 of them are data nodes) and large scale of 100 nodes (20 of them are data nodes). The sensor nodes are uniformly distributed in a region of $1000m \times 1000m$. Each data node generates some number of data packets, each of 512B, that to be offloaded into the network. For initial energy levels, we consider both *varying model*, where different sensors have different initial energy, and *uniform model*, where all sensor nodes have the same initial energy level. Transmission range is 250m. In all plots, each data point is an average over ten runs, in each of which a different sensor network is generated; the error bars indicate 95%

confidence intervals. For fair comparisons, in each run different algorithms use the same input including network topology, initial energy of each node E_i, the data nodes and their numbers of data packets d_i, and the storage nodes and their storage capacities m_j. For each sensor network, we first run feasibility checking – if not feasible, we generate another one and check again. All QP, MCF ILP and feasibility checking using maximum flow ILP are implemented in CPLEX [2].



(a) Data resilience level. (b) Total Energy Cost.

Fig. 4. Small-scale comparison by varying $d_i, m_j = 5$.



(a) Data resilience level. (b) Total Energy Cost.

Fig. 5. Small-scale comparison by varying $m_j, d_i = 10$.

7.1 Small-scale Comparison

As QP takes long time to run, we compare them in small scale of 50 nodes with 10 data nodes. The initial energy levels are random numbers in $[2000\mu J, 4000\mu J]$. Fig. 4(a) varies d_i from 5, 10, 15, to 20 while setting m_j as 5. It shows that with the increase of d_i, the DRLs achieved by all algorithms increase, as more data packets are now stored in storage nodes. QP always achieves slightly higher DRLs than MCF, while MCF higher than Network most of the time. As QP is optimal and all three perform close, this demonstrates the efficacy of all three algorithms in achieving data resilience. Fig. 4(b) shows the total energy consumptions of three algorithms, among which MCF has the smallest. QP costs the most energy, though, as it sometimes detours (instead of the shortest energy path) from sources to destinations in order to achieve higher DRLs. Fig. 5 varies m_j while fixing d_i = 10. It shows again that QP achieves highest DRLs with the cost of energy consumption.

TABLE II. Investigating Tolerance Gap of The QP in CPLEX

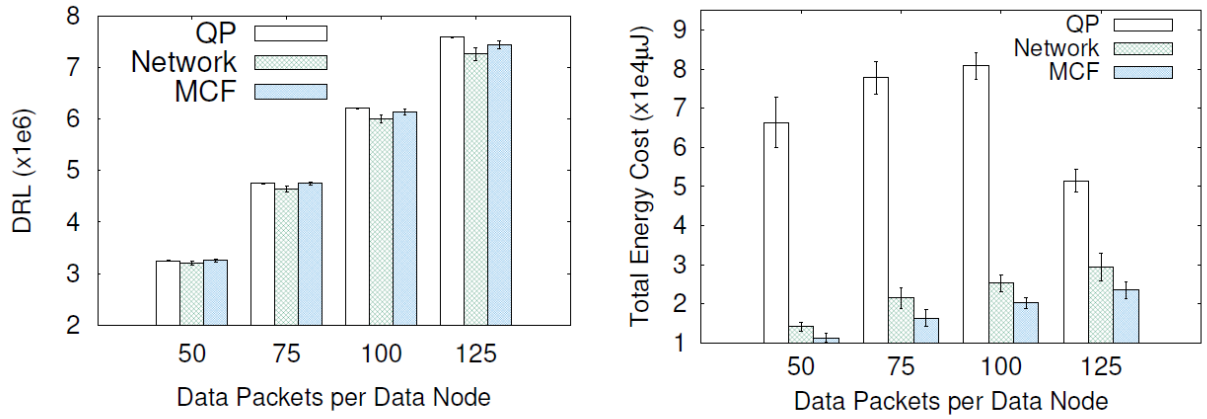
Tolerance Gap (%)	2	3	5	10	30
Execution time (sec)	2014.91	826.3	11.98	13.44	15.92
	788.8	183.53	8.27	7.89	7.98
	1034.27	160.98	22.64	22.8	28.32

7.2 Large-scale Comparison

Next, we compare the algorithms in larger scale of 100 sensors, 20 of them are data nodes. As CPLEX [2] cannot finish computing an instance of QP after more than 13 hours, we decide to resort to below technique.

Gap Tolerance of the QP. To improve the time efficiency of QP, CPLEX [2] can be parameterized using a percentage value called *gap tolerance*. CPLEX stops once it finds a feasible solution within this percent of optimal. We thus investigate the tradeoff between time-efficiency

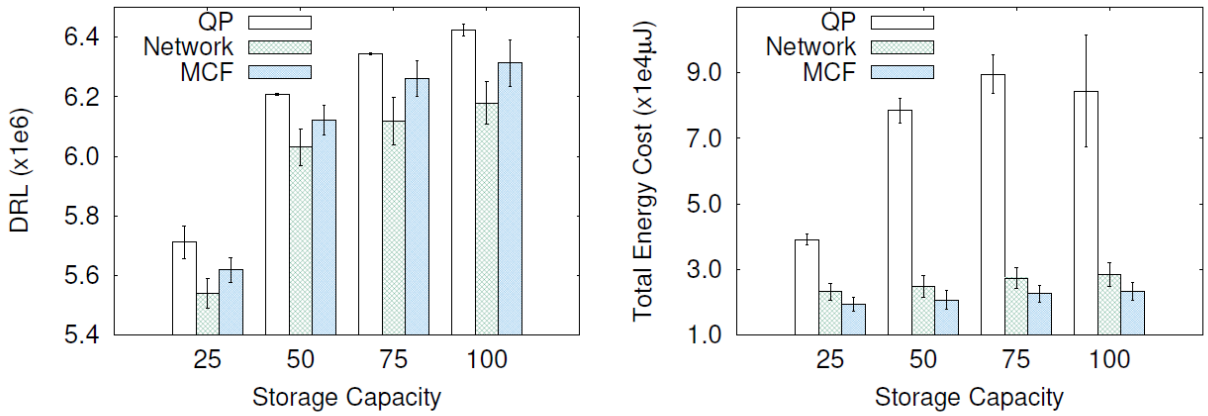
and solution quality of different gap tolerances. Table II records the CPLEX execution time for different tolerance gaps between 2% and 30%, for three randomly generated networks. As 2% can take more than half an hour, we choose the next value of 3% as the gap tolerance for the QP; i.e., for the rest comparisons the QP always achieves at least 97% of the optimal DRLs.



(a) Data resilience level. (b) Total Energy Cost.

Fig. 6. Large-scale comparison by varying $d_i, m_j = 50$.

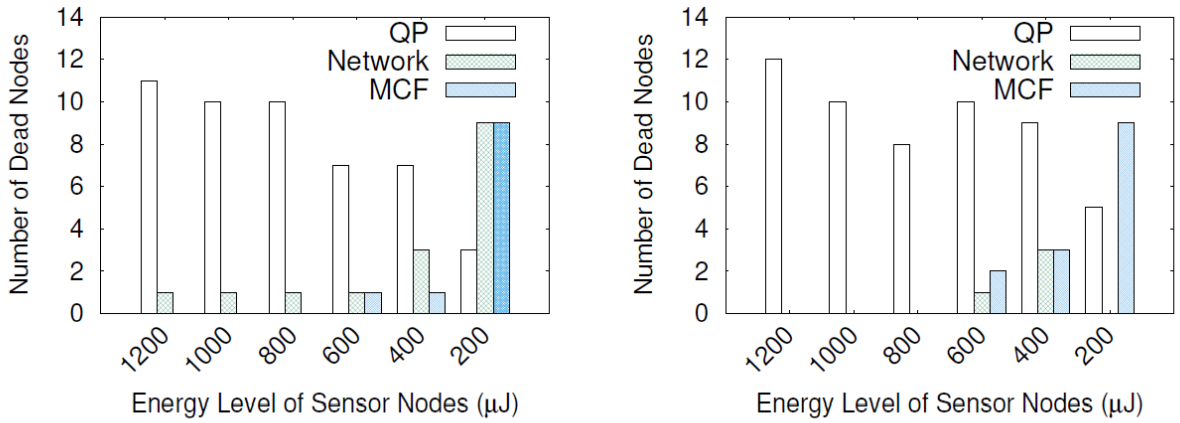
Fig. 6 compare the three algorithms by varying d_i from 50, 75, 100, to 125 with $m_j = 50$ and $E_i = 2500\mu J$. We observe that even with fault tolerance, QP still outperforms the Network and MCF in DRLs. However, Fig. 6 (b) shows the energy cost of QP is much larger than those of the other two, as its focus is on high DRLs and not on the energy costs to achieve them. It also shows that the energy cost of QP decreases when d_i increases from 100 to 125. This is rather counter-intuitive, as offloading more data packets should cost more energy. Our conjecture for QP is that if there are multiple routes to offload a data packet without affecting the DRL maximization, it randomly chooses one as long as the gap tolerance level is met. When the network has more data packets to store, however, choosing such a random path could negatively affect the DRL maximization. As such, the QP begins choosing more energy-efficient offloading paths thus decreasing the energy cost.



(a) Data resilience level. (b) Total Energy Cost.

Fig. 7. Large-scale comparison by varying $m_j, d_i = 100$.

Fig. 7 varies m_j and shows that the performance differences of DRLs achieved by different algorithms seem to increase when increasing the storage capacity. As high energy nodes have more spaces to store data packets, the QP, being optimal, does a better job of utilizing the available spaces in order to maximize the DRL.



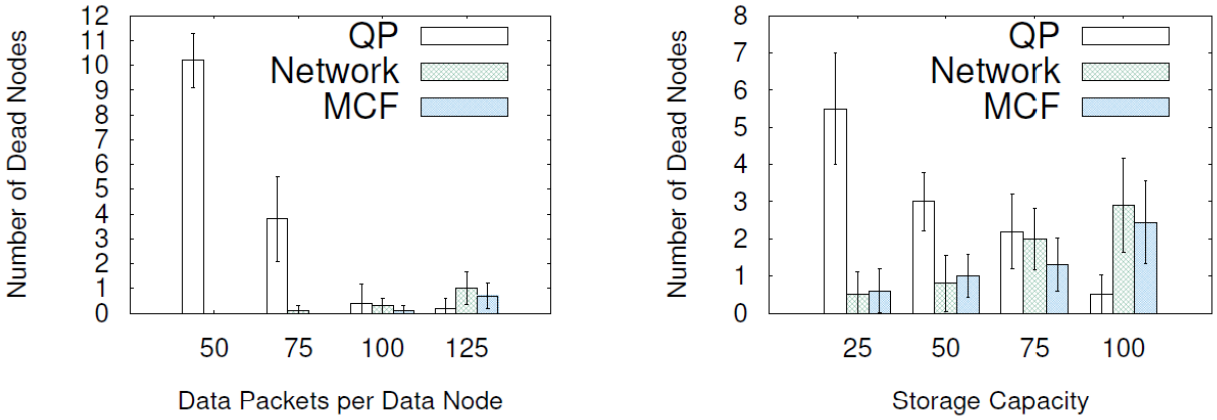
(a) $m_j = 50$. (b) $m_j = 13$.

Fig. 8. Large-scale comparison by varying E_i.

7.3 Investigating Fault-Tolerance

Finally, we investigate the fault-tolerance capability of the three algorithms by finding the number of dead nodes (i.e., nodes with depleted energy). It randomly generates one sensor network

of 100 nodes, 20 of them are data nodes with d_j = 50. It starts by setting the initial energy levels E_i of all the nodes as $1200\mu J$, gradually decreases them, and records the number of dead nodes along the way for three algorithms. It stops until at least one of them fails to offload all the data packets. Fig. 8(a) sets m_j as 50 and shows all algorithms can tolerate up to around 10 node failures. However, as QP focuses more on DRLs and less on energy costs, it incurs more dead nodes than the other two most of the time. Fig 8(b) decreases m_j to 13, at which the network is almost full after data offloading, and shows that they can tolerate up to 6 and 8 node failures respectively before data loss occurs (note that when $E_i = 200\mu J$, Network cannot offload all the data packets). With the decrease of m_j, more storage nodes participate in the data offloading process (either storing or relaying), thus consuming more energy compared to when m_j is larger. Consequently, it allows a smaller number of dead nodes to take place before failing to offload all the data packets.



(a) Varying d_i. $m_j = 50$. (b) Varying m_j. $d_i = 100$.

Fig. 9. Fault-tolerance of three algorithms at $E_i = 1200\mu J$.

Last, Fig. 9(a) and (b) study the fault-tolerance of algorithms at $E_i = 1200\mu J$ by varying d_i and m_j respectively. It is interesting to notice that with the increase of m_j, the number of dead nodes increases for both Network and MCF while decreasing for QP. For Network and MCF, as the number of destination nodes gets smaller with increase of m_j, less number of them participated

in the data offloading process, depleting their energies more quickly. For QP, with the increase of m_j it can distribute data packets to nodes more evenly, thus reducing the number of dead nodes.

8. CONCLUSION AND FUTURE WORK

We solved a new algorithmic problem called DRE^2 that achieved maximum data resilience inside sensor networks. It uniquely arises from emerging sensor network applications that are deployed in extreme environments. We designed a QP-based optimal algorithm and two time- and energy-efficient heuristic algorithms viz. Network and MCF. We also solved the feasibility problem of DRE^2 by designing a maximum flow-based algorithm. Although sensor network research has been around for more than two decades, we believe that data resilience in our sensor network model for emerging applications has not been thoroughly addressed in existing literature. We uncovered a generalized edge capacity constraint model, wherein the consumed capacity on an edge is the linear combination of its flows. This generalizes the well-accepted edge capacity constraint in traditional network flows. As a future work, we will study if our heuristics can provide any performance guarantees in terms of DRLs. We plan to focus on a few specific topologies such as stars and trees, and investigate if the optimal and time-efficient algorithms exist.

9. REFERENCES

[1] Cgal 5.0.2 - linear and quadratic programming solver. https://doc.cgal.org/latest/QP_solver/index.html.

[2] Ibm cplex optimizer. https://www.ibm.com/analytics/cplex-optimizer.

[3] G. Aathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Ultra-low power data storage for sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 5(4), 2009.

[4] M. Albano and J. Gao. Resilient data-centric storage in wireless ad-hoc sensor networks. In *Proc. of ALGOSENSOR*, 2010.

[5] B. Alhakami, B. Tang, J. Han, and M. Beheshti. Dao-r: Integrating data aggregation and offloading in sensor networks via data replication. *International Journal of Sensor Networks*, 29(2):134 – 146, 2019.

[6] Y. Chen and B. Tang. Data preservation in base station-less sensor networks: A game theoretic approach. In *Proc. of the 6th EAI International Conference on Game Theory for Networks (GameNets 2016)*.

[7] G. Citovsky, J. Gao, J. S. B. Mitchell, and J. Zeng. Exact and approximation algorithms for data mule scheduling in a sensor network. In *Proc. of ALGOSENSORS 2015*.

[8] T. Cormen, C. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second ed.)*. MIT Press and McGraw–Hill Press, 2009.

[9] R. W. L. Coutinho, A. Boukerche, and S. Guercin. Performance evaluation of candidate set selection procedures in underwater sensor networks. In *Proc. of IEEE ICC 2019*.

[10] L. Doherty, K. S. J. pister, and L. El Ghaoui. Convex position estimation in wireless sensor networks. In *Proc. of IEEE INFOCOM 2001*, 2001.

[11] S. Edwards, T. Murray, T. O'Farrell, I. C. Rutt, P. Loskot, I. Martin, N. Selmes, R. Aspey, T. James, S. L. Bevan, and T. Bauge. A´ High-Resolution Sensor Network for Monitoring Glacier Dynamics. *IEEE SENSORS JOURNAL*, 14(11):3926–3931, 2013.

[12] C. A. Floudas and V. Visweswaran. Quadratic optimization. *Nonconvex Optimization and Its Applications*, 2:217–269, 1995.

[13] M. Frank and P. Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3:95–110, 1956. Web. 4 June 2015.

[14] D. Ganesan, R. Govindan, S. Shenker, and D. Estrin. Highly-resilient, energy-efficient multipath routing in wireless sensor networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(4):11–25, October 2001.

[15] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[16] A. Ghose, J. Grossklags, and J. Chuang. Resilient data-centric storage in wireless ad-hoc sensor networks. In *Proc. of MDM*, 2003.

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

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

[19] X. Hou, Z. Sumpter, L. Burson, X., and B. Tang. Maximizing data preservation in intermittently connected sensor networks. In *Proc. of IEEE MASS 2012*, pages 448–452.

[20] Y. Huang, J. F. Mart´ınez, J. Sendra, and L. Lopez. Resilient wireless sensor networks using topology control: A review. *Sensors (Basel)*, 15(10):24735–24770, 2015.

[21] A. Kamra, J. Feldman, V. Misra, and D. Rubenstein. Growth codes: Maximizing sensor network data persistence. In *Proc. of SIGCOMM*, 2006.

[22] J. Lee, W. Chung, and E. Kim. A new range-free localization method using quadratic programming. *Computer Communications*, 34(8):998–1010, 2011.

[23] F. Liu, M. Lin, Y. Hu, C. Luo, and F. Wu. Design and analysis of compressive data persistence in large-scale wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 26(10):2685–2698, 2015.

[24] L. Liu and R. Wang. Message dissemination for throughput optimization in storage-limited opportunistic underwater sensor networks. In *Proc. of SECON 2016*.

[25] L. Luo, C. Huang, T. Abdelzaher, and J. Stankovic. Envirostore: A cooperative storage system for disconnected operation in sensor networks. In *Proc. of INFOCOM 2007*.

[26] D. E. Phillips, M. Moazzami, G. Xing, and J. M. Lees. A sensor network for real-time volcano tomography: System design and deployment. In *Proc. of IEEE ICCCN 2017*.

[27] D. Puccinelli and M. Haenggi. Reliable data delivery in large-scale low-power sensor networks. *ACM Trans. Sen. Netw.*, 6(4):28:1–28:41, 2010.

[28] M. Rahmati and D. Pompili. Uwsvc: Scalable video coding transmission for in-network underwater imagery analysis. In *Proc. of IEEE MASS 2019*.

[29] K. A. Ravindra, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[30] R. Sugihara and R. K. Gupta. Path planning of data mules in sensor networks. *ACM Trans. Sen. Netw.*, 8(1):1:1–1:27, 2011.

[31] Z. Sun and I. F. Akyildiz. On capacity of magnetic induction-based wireless underground sensor networks. In *Proc. of INFOCOM*, 2012.

[32] Rui Tan, Guoliang Xin, Jinzhu Chen, Wen-Zhan Song, and Renjie Huang. Fusion-based volcanic earthquake detection and timing in wireless sensor networks. *ACM Transaction on Sensor Networks (ACM TOSN)*, 9, 2013.

[33] B. Tang. *dao*^{2}: Overcoming overall storage overflow in intermittently connected sensor networks. In *Proc. of IEEE INFOCOM 2018*.

[34] B. Tang, N. Jaggi, and M. Takahashi. Achieving data k-availability in intermittently connected sensor networks. In *Proc. of IEEE ICCCN 2014*.

[35] B. Tang, N. Jaggi, H. Wu, and R. Kurkal. Energy efficient data redistribution in sensor networks. *ACM Transactions on Sensor Networks*, 9(2):1–28, May 2013.

[36] A. Tinka, I. Strub, Q. Wu, and A. M. Bayen. Quadratic programming based data assimilation with passive drifting sensors for shallow water flows. In *Proc. of IEEE Conference on Decision and Control (CDC)*, 2009.

[37] L. Wang, Y. Yang, D. K. Noh, H. L., T. Abdelzaher, M. Ward, and J. Liu. Adaptsens: An adaptive data collection and storage service for solar-powered sensor networks. In *Proc. of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*.

[38] X. Xu, H. Zhang, T. Li, and L. Zhang. Achieving resilient data availability in wireless sensor networks. In *Proc. of IEEE ICC 2019 Workshops*.

[39] X. Xue, X. Hou, B. Tang, and R. Bagai. Data preservation in intermittently connected sensor networks with data priorities. In *Proc. of IEEE SECON 2013*, pages 65–73.

[40] Yong Yang, Lili Wang, Dong Kun Noh, Hieu Khac Le, and Tarek F. Abdelzaher. Solarstore: enhancing data reliability in solar-powered storage-centric sensor networks. In *Proc. of MobiSys 2009*, pages 333–346, 2009.

[41] M. Z. Zamalloa and B. Krishnamachari. An analysis of unreliability and asymmetry in low-power wireless links. *ACM Transactions on Sensor Networks*, 3(2):1277–1280, 2007.

10. APPENDIX

10.1 Sensor Network Simulation

```
/**
 * This program will generate a sensor network graph
 * and observe the graph's status (cost, data item's amount received / send / saved, dead or live).
 *
 * Different constraints and algorithm used to complete the task depends on users' input.
 *
 * @author  Shang-Lin Hsu
 * @since   2018-09-09
 */

public class SensorNetworkCPLX {

   static Random rand = new Random();
   static XSSFWorkbook energyworkbook = new XSSFWorkbook();
   static XSSFSheet energysheet = energyworkbook.createSheet("EnergyCostData");
   static Map<Integer, Axis> nodes = 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>();
  Stack<Integer> s = new Stack<Integer>();
  static Map<String, Link> links = new HashMap<String, Link>();
  static Map<String, Link> linkstest = new HashMap<String, Link>();
  static HashMap<Integer, List<Integer>> close = new HashMap<>();

   static int minCapacity;
   static int biconnectcounter = 1;
   static int[] dataGens;
   static int[] storageNodes;
   static int numberOfDG;
   static int numberOfDataItemsPerDG;
   static int numberOfStoragePerSN;
   static int numberOfNodes;
   // 1 = milli, 1000 = nero
  static int baseUnit = 1;

  public static void main(String[] args) throws IOException, IloException {
    Scanner scan = new Scanner(System.in);
    System.out.println("The width (meters) is set to:");
//    double width = scan.nextDouble();
      double width = 1000.0;
      System.out.println(width);

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

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

```
        //numberOfNodes = 100;

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

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

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

    System.out.println("Data storage per storage node is set to: ");
    numberOfStoragePerSN = scan.nextInt();
    // CHANGE
    //numberOfStoragePerSN = 25;

    System.out.println("Please enter the initial energy capacity (micro-J ex: 2500):");
    System.out.println("**Note: The energy will be a random amount between your input and your input
+ 1000.");
    // max energy calculation use nano-J (leave out decimal point to reduce the precess time)
    minCapacity = scan.nextInt() * baseUnit;

    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: " + numberOfstorage);

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

    SensorNetworkCPLX sensor = new SensorNetworkCPLX();

    // random generate the graph
    populateNodes(numberOfNodes, width, height);

    System.out.println("\nNode List:");
```

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

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

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();

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<>(linkstest);

Map<Integer, Double> sortMap = new HashMap<>();

// ---------- set to random energy base on user input ----------
for (Map.Entry<Integer, Set<Integer>> node : adjacencyList1.entrySet()) {
  int temp = (rand.nextInt(1000 * baseUnit)) + minCapacity;
  int i = node.getKey();
  if (i <= numberOfDG) {
    for (Link link : treeMap.values()) {
      // if the node is data generator, the energy will be max (minCapacity + max random number)
      if (i == link.getEdge().getHead()) {
        link.setEnergy(minCapacity + (1000 * baseUnit));
```

```
            sortMap.put(link.getEdge().getHead(), (double) minCapacity + (1000 * baseUnit));
          }
        }
      } else {
        for (Link link : treeMap.values()) {
          if (i == link.getEdge().getHead()) {
            link.setEnergy(temp);
            sortMap.put(link.getEdge().getHead(), (double) temp);
          }
        }
      }
    }

    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");
        }
      }
    }

    // write original energy to file
//    File EnergyFile = new File("Edge_cost_Original.txt");
//    BufferedWriter writer_energy = new BufferedWriter(new FileWriter(EnergyFile));
//    writer_energy.write(totalenergycost.toString());
//    writer_energy.close();

    // file for energy sorting

    PriorityQueue<Map.Entry<Integer, Double>> copyEnergy = new PriorityQueue<>((a,b) ->
        b.getValue().intValue() - a.getValue().intValue());
    for (Map.Entry<Integer, Double> entry : sortMap.entrySet()) {
      copyEnergy.offer(entry);
    }
    while (!copyEnergy.isEmpty()) {
      copyEnergy.poll();
    }

    // create first row
    int rowcounter = 0;
    Row energyrow = energysheet.createRow(rowcounter++);

    // name for cols
    String[] row = new String[]{"RemoveNode", "C_{V-{i}}", "C_V", "dataitems", "CPLEX_energycost",
        "CPLEX_Obj", "Network_base_energycost", "Network_base_data_resilience",
"MinCostFlow_energycost",
        "MinCostFlow_resilience", "CPLEX_Dead", "Algo_Dead", "MinCost_Dead", "Base_energy_capacity",
        "Base_storage_capacity", "Gap_tolerance(%)"};
    // write
    for (int i = 0; i < row.length; i++) {
```

```
        Cell energycell = energyrow.createCell(i);
        energycell.setCellValue(row[i]);
    }

    System.out.println("Select the data you want to generate: ");
    System.out.println("0 = observe cplex gap tolerance's difference (will require user to input gap tolerance 5 times)");
    System.out.println("1 = data priority (Linear Programming): gradually increase data generator's amount to observe data priority solutions");
    System.out.println("2 = data resilience (Quadratic Programming): change storage (every run + 25), fix data item's amount");
    System.out.println("3 = data resilience (Quadratic Programming): fix storage, change data item's amount (every time + DG's amount * 5)");
    System.out.println("4 = data resilience (Quadratic Programming): change storage (user defines common difference)");
    System.out.println("5 = data resilience (Quadratic Programming): change data item's amount (user defines common difference **Difference = DG's amount * user defined value)");
    System.out.println("6 = data resilience (Quadratic Programming): run twice different storage, decrease energy gradually to observe fault tolerance");
    System.out.println("7 = data resilience (Quadratic Programming): run twice different data items, observe how much data items sent to the maximize DRL");

    // select method
    int runMethod = scan.nextInt();
    if (runMethod == 0) {
      System.out.println("This program will run 5 times to generate 5 different gap tolerance.");
      for (int i = 0; i < 5; i++) {
        System.out.println("Input the gap tolerance (Percentage e.g: 0.8% = 0.008, 1% = 0.01):");
        double gapTolerance = scan.nextDouble();

        // create new row to store nre data
        energyrow = energysheet.createRow(rowcounter++);

        /* ---------------------------------- QP ----------------------------------------*/
        QP_cplex qp_method = new QP_cplex(dataGens, storageNodes, numberOfDataItemsPerDG,
numberOfStoragePerSN, treeMap ,adjacencyList1 ,close);
        qp_method.solve(gapTolerance, "original");

        Cell cell3 = energyrow.createCell(3);
        cell3.setCellValue(qp_method.getDataSend());
        Cell cell4 = energyrow.createCell(4);
        cell4.setCellValue(qp_method.getEnergyCost(true));
        Cell cell5 = energyrow.createCell(5);
        cell5.setCellValue(qp_method.getCplexObj());

        ArrayList<Integer> deadNodes = qp_method.getDeadNodes();
        if (deadNodes.size() > 0) {
          Cell CPLEXDead = energyrow.createCell(10);
          CPLEXDead.setCellValue(Arrays.toString(deadNodes.toArray()));
        }

        /* ------------------------------ network base ---------------------------------- */
        // run Algorithm
        NetworkBase networkAlgo = new NetworkBase(dataGens, storageNodes, numberOfDataItemsPerDG,
numberOfStoragePerSN, treeMap);
```

```
        boolean algOk = networkAlgo.solve();

        Cell cell6 = energyrow.createCell(6);
        cell6.setCellValue(networkAlgo.getTotalEnergyCost());
        Cell cell7 = energyrow.createCell(7);
        cell7.setCellValue(networkAlgo.getDataResilience());

        System.out.println();
        if (!algOk) {
          System.out.println("have problem: network base");
        }
        // check dead
        PriorityQueue<Integer> deadAlgo = networkAlgo.getDeadnodes();
        if (!deadAlgo.isEmpty()) {
          ArrayList<Integer> deadAlgoNodes = new ArrayList<>();
          while (!deadAlgo.isEmpty()) {
            deadAlgoNodes.add(deadAlgo.poll());
          }
          Cell AlgoDead = energyrow.createCell(11);
          AlgoDead.setCellValue(Arrays.toString(deadAlgoNodes.toArray()));
        }

        /* ---------------------------------- MCF ILP ------------------------------------ */
        boolean[] sendOrNot = new boolean[numberOfNodes + 1];
        int tempData = 0;

        // sort the energy for the mincost usage
        PriorityQueue<Map.Entry<Integer, Double>> sortEnergy = new PriorityQueue<>((a, b) ->
            b.getValue().intValue() - a.getValue().intValue());
        for (Map.Entry<Integer, Double> entry : sortMap.entrySet()) {
          sortEnergy.offer(entry);
        }

        Arrays.fill(sendOrNot, false);
        // if the energy is high, send it
        while (!sortEnergy.isEmpty() && tempData < numberOfDG * numberOfDataItemsPerDG) {
          if (sortEnergy.peek().getKey() > numberOfDG) {
            int node = sortEnergy.peek().getKey();
            tempData += numberOfStoragePerSN;
            sendOrNot[node] = true;
          }
          sortEnergy.poll();
        }

        // run MInCost
        MinCostILP minCost = new MinCostILP(dataGens, storageNodes, minCapacity,
numberOfDataItemsPerDG,
            numberOfStoragePerSN, treeMap, adjacencyList1, close);
        minCost.solve("original", sendOrNot);

        Cell minCostEnrgy = energyrow.createCell(8);
        minCostEnrgy.setCellValue(minCost.getILPObj());
        Cell minCostResilience = energyrow.createCell(9);
        minCostResilience.setCellValue(minCost.getResilience());
```

```
        ArrayList<Integer> mindeadNodes = minCost.getDeadNodes();
        if (mindeadNodes.size() > 0) {
          Cell MinCostDead = energyrow.createCell(12);
          MinCostDead.setCellValue(Arrays.toString(mindeadNodes.toArray()));
        }

        /* ------------------------  input information ---------------------------- */
        Cell baseEnergy = energyrow.createCell(13);
        baseEnergy.setCellValue(minCapacity);
        Cell baseStorage = energyrow.createCell(14);
        baseStorage.setCellValue(numberOfStoragePerSN);
        Cell BaseGap = energyrow.createCell(15);
        BaseGap.setCellValue(gapTolerance);
      }
    }  else if (runMethod == 1) {
      System.out.println("Running priority formulation, please input the initial energy:");
      System.out.println("Still Under construction....");
      System.out.println("This part is preserved for further data priority research");

    } else if (runMethod == 2 || runMethod == 3) {
      System.out.println("Input the gap tolerance (Percentage e.g: 0.8% = 0.008, 1% = 0.01):");
      double gapTolerance = scan.nextDouble();
      for (int i = 0; i < 4; i++) {
        if (runMethod == 2) {
          System.out.println("Now running storage capacity: " + numberOfStoragePerSN);
        } else {
          System.out.println("Now running total data items: " + numberOfDataItemsPerDG);
        }

        // create new row to store nre data
        energyrow = energysheet.createRow(rowcounter++);

        /* ------------------------------ QP part ------------------------------------ */
        // CPLEX_QP
        QP_cplex qp_method = new QP_cplex(dataGens, storageNodes, numberOfDataItemsPerDG,
numberOfStoragePerSN, treeMap ,adjacencyList1 ,close);
        qp_method.solve(gapTolerance, "original");

        // data items sent
        Cell cell3 = energyrow.createCell(3);
        cell3.setCellValue(qp_method.getDataSend());

        Cell cell4 = energyrow.createCell(4);
        cell4.setCellValue(qp_method.getEnergyCost(true));
        Cell cell5 = energyrow.createCell(5);
        cell5.setCellValue(qp_method.getCplexObj());

        ArrayList<Integer> deadNodes = qp_method.getDeadNodes();
        if (deadNodes.size() > 0) {
          Cell CPLEXDead = energyrow.createCell(10);
          CPLEXDead.setCellValue(Arrays.toString(deadNodes.toArray()));
        }

        /* ------------------------ network base part ------------------------------ */
        // run Algorithm
```

```
        NetworkBase networkAlgo = new NetworkBase(dataGens, storageNodes, numberOfDataItemsPerDG,
numberOfStoragePerSN, treeMap);
        boolean algOk = networkAlgo.solve();

        Cell cell6 = energyrow.createCell(6);
        cell6.setCellValue(networkAlgo.getTotalEnergyCost());
        Cell cell7 = energyrow.createCell(7);
        cell7.setCellValue(networkAlgo.getDataResilience());

        System.out.println();
        if (!algOk) {
          System.out.println("have problem: network base");
        }
        // check dead
        PriorityQueue<Integer> deadAlgo = networkAlgo.getDeadnodes();
        if (!deadAlgo.isEmpty()) {
          ArrayList<Integer> deadAlgoNodes = new ArrayList<>();
          while (!deadAlgo.isEmpty()) {
            deadAlgoNodes.add(deadAlgo.poll());
          }
          Cell AlgoDead = energyrow.createCell(11);
          AlgoDead.setCellValue(Arrays.toString(deadAlgoNodes.toArray()));
        }

        /* ---------------------------------- MCF ILP part ---------------------------------- */
        // decide which node to send
        boolean[] sendOrNot = new boolean[numberOfNodes + 1];
        int tempData = 0;

        // sort the energy for the mincost usage
        PriorityQueue<Map.Entry<Integer, Double>> sortEnergy = new PriorityQueue<>((a, b) ->
b.getValue().intValue() - a.getValue().intValue());
        for (Map.Entry<Integer, Double> entry : sortMap.entrySet()) {
          sortEnergy.offer(entry);
        }

        Arrays.fill(sendOrNot, false);
        // if the energy is high, send it
        while (!sortEnergy.isEmpty() && tempData < numberOfDG * numberOfDataItemsPerDG) {
          if (sortEnergy.peek().getKey() > numberOfDG) {
            int node = sortEnergy.peek().getKey();
            tempData += numberOfStoragePerSN;
            sendOrNot[node] = true;
          }
          sortEnergy.poll();
        }

        // run MInCost
        MinCostILP minCost = new MinCostILP(dataGens, storageNodes, minCapacity,
numberOfDataItemsPerDG,
            numberOfStoragePerSN, treeMap, adjacencyList1, close);
        minCost.solve("original", sendOrNot);

        // energy cost and resilience level
        Cell minCostEnrgy = energyrow.createCell(8);
```

```
        minCostEnrgy.setCellValue(minCost.getILPObj());
        Cell minCostResilience = energyrow.createCell(9);
        minCostResilience.setCellValue(minCost.getResilience());

        ArrayList<Integer> mindeadNodes = minCost.getDeadNodes();
        if (mindeadNodes.size() > 0) {
          Cell MinCostDead = energyrow.createCell(12);
          MinCostDead.setCellValue(Arrays.toString(mindeadNodes.toArray()));
        }

        /*------------------------- extra information ----------------------*/
        // base energy / storage / gap
        Cell baseEnergy = energyrow.createCell(13);
        baseEnergy.setCellValue(minCapacity);
        Cell baseStorage = energyrow.createCell(14);
        baseStorage.setCellValue(numberOfStoragePerSN);
        Cell BaseGap = energyrow.createCell(15);
        BaseGap.setCellValue(gapTolerance);

        if (runMethod == 2) {
          numberOfStoragePerSN += 25;
        } else {
          numberOfDataItemsPerDG += 25;
        }
      }
    } else if (runMethod == 4) {
      System.out.println("the program will run 4 times. Please set the common difference for each
run:");
      int commonDifference = scan.nextInt();
      double gapTolerance = 0.03;
      for (int i = 0; i < 4; i++) {
        System.out.println("Now running storage capacity: " + numberOfStoragePerSN);

        // create new row to store nre data
        energyrow = energysheet.createRow(rowcounter++);

        /* ------------------------------ QP part ---------------------------------- */
        // CPLEX_QP
        QP_cplex qp_method = new QP_cplex(dataGens, storageNodes, numberOfDataItemsPerDG,
            numberOfStoragePerSN, treeMap ,adjacencyList1 ,close);
        qp_method.solve(gapTolerance, "original");


        // data items sent
        Cell cell3 = energyrow.createCell(3);
        cell3.setCellValue(qp_method.getDataSend());

        Cell cell4 = energyrow.createCell(4);
        cell4.setCellValue(qp_method.getEnergyCost(true));
        Cell cell5 = energyrow.createCell(5);
        cell5.setCellValue(qp_method.getCplexObj());

        ArrayList<Integer> deadNodes = qp_method.getDeadNodes();
        if (deadNodes.size() > 0) {
          Cell CPLEXDead = energyrow.createCell(10);
```

```
            CPLEXDead.setCellValue(Arrays.toString(deadNodes.toArray()));
        }

        /* ---------------------------- network base part ------------------------------ */
        // run Algorithm
        NetworkBase networkAlgo = new NetworkBase(dataGens, storageNodes, numberOfDataItemsPerDG,
numberOfStoragePerSN, treeMap);
        boolean algOk = networkAlgo.solve();

        Cell cell6 = energyrow.createCell(6);
        cell6.setCellValue(networkAlgo.getTotalEnergyCost());
        Cell cell7 = energyrow.createCell(7);
        cell7.setCellValue(networkAlgo.getDataResilience());

        System.out.println();
        if (!algOk) {
            System.out.println("have problem: network base");
        }
        // check dead
        PriorityQueue<Integer> deadAlgo = networkAlgo.getDeadnodes();
        if (!deadAlgo.isEmpty()) {
            ArrayList<Integer> deadAlgoNodes = new ArrayList<>();
            while (!deadAlgo.isEmpty()) {
                deadAlgoNodes.add(deadAlgo.poll());
            }
            Cell AlgoDead = energyrow.createCell(11);
            AlgoDead.setCellValue(Arrays.toString(deadAlgoNodes.toArray()));
        }

        /* ---------------------------------- MCF ILP part ---------------------------------- */
        // decide which node to send
        boolean[] sendOrNot = new boolean[numberOfNodes + 1];
        int tempData = 0;

        // sort the energy for the mincost usage
        PriorityQueue<Map.Entry<Integer, Double>> sortEnergy = new PriorityQueue<>((a, b) ->
b.getValue().intValue() - a.getValue().intValue());
        for (Map.Entry<Integer, Double> entry : sortMap.entrySet()) {
            sortEnergy.offer(entry);
        }

        Arrays.fill(sendOrNot, false);
        // if the energy is high, send it
        while (!sortEnergy.isEmpty() && tempData < numberOfDG * numberOfDataItemsPerDG) {
            if (sortEnergy.peek().getKey() > numberOfDG) {
                int node = sortEnergy.peek().getKey();
                tempData += numberOfStoragePerSN;
                sendOrNot[node] = true;
            }
            sortEnergy.poll();
        }

        // run MInCost
        MinCostILP minCost = new MinCostILP(dataGens, storageNodes, minCapacity,
numberOfDataItemsPerDG, numberOfStoragePerSN, treeMap, adjacencyList1, close);
```

```
        minCost.solve("original", sendOrNot);

        // energy cost and resilience level
        Cell minCostEnrgy = energyrow.createCell(8);
        minCostEnrgy.setCellValue(minCost.getILPObj());
        Cell minCostResilience = energyrow.createCell(9);
        minCostResilience.setCellValue(minCost.getResilience());

        ArrayList<Integer> mindeadNodes = minCost.getDeadNodes();
        if (mindeadNodes.size() > 0) {
          Cell MinCostDead = energyrow.createCell(12);
          MinCostDead.setCellValue(Arrays.toString(mindeadNodes.toArray()));
        }

        /*------------------------- extra information ----------------------*/
        // base energy / storage / gap
        Cell baseEnergy = energyrow.createCell(13);
        baseEnergy.setCellValue(minCapacity);
        Cell baseStorage = energyrow.createCell(14);
        baseStorage.setCellValue(numberOfStoragePerSN);
        Cell BaseGap = energyrow.createCell(15);
        BaseGap.setCellValue(gapTolerance);

        // each run add the common difference
        numberOfStoragePerSN += commonDifference;
      }
    } else if (runMethod == 5) {
      System.out.println("the program will run 4 times. Please set the common difference for each
run:");
      int commonDifference = scan.nextInt();
      double gapTolerance = 0.03;
      for (int i = 0; i < 4; i++) {
        System.out.println("Now running data items per DG: " + numberOfDataItemsPerDG);

        // create new row to store nre data
        energyrow = energysheet.createRow(rowcounter++);

        /* ------------------------------ QP part ---------------------------------- */
        // CPLEX_QP
        QP_cplex qp_method = new QP_cplex(dataGens, storageNodes, numberOfDataItemsPerDG,
            numberOfStoragePerSN, treeMap ,adjacencyList1 ,close);
        qp_method.solve(gapTolerance, "original");

        // data items sent
        Cell cell3 = energyrow.createCell(3);
        cell3.setCellValue(qp_method.getDataSend());

        Cell cell4 = energyrow.createCell(4);
        cell4.setCellValue(qp_method.getEnergyCost(true));
        Cell cell5 = energyrow.createCell(5);
        cell5.setCellValue(qp_method.getCplexObj());

        ArrayList<Integer> deadNodes = qp_method.getDeadNodes();
        if (deadNodes.size() > 0) {
          Cell CPLEXDead = energyrow.createCell(10);
```

```
            CPLEXDead.setCellValue(Arrays.toString(deadNodes.toArray()));
        }

        /* ---------------------------- network base part ------------------------------ */
        // run Algorithm
        NetworkBase networkAlgo = new NetworkBase(dataGens, storageNodes, numberOfDataItemsPerDG,
numberOfStoragePerSN, treeMap);
        boolean algOk = networkAlgo.solve();

        Cell cell6 = energyrow.createCell(6);
        cell6.setCellValue(networkAlgo.getTotalEnergyCost());
        Cell cell7 = energyrow.createCell(7);
        cell7.setCellValue(networkAlgo.getDataResilience());

        System.out.println();
        if (!algOk) {
            System.out.println("have problem: network base");
        }
        // check dead
        PriorityQueue<Integer> deadAlgo = networkAlgo.getDeadnodes();
        if (!deadAlgo.isEmpty()) {
            ArrayList<Integer> deadAlgoNodes = new ArrayList<>();
            while (!deadAlgo.isEmpty()) {
                deadAlgoNodes.add(deadAlgo.poll());
            }
            Cell AlgoDead = energyrow.createCell(11);
            AlgoDead.setCellValue(Arrays.toString(deadAlgoNodes.toArray()));
        }

        /* ---------------------------------- MCF ILP part ---------------------------------- */
        // decide which node to send
        boolean[] sendOrNot = new boolean[numberOfNodes + 1];
        int tempData = 0;

        // sort the energy for the mincost usage
        PriorityQueue<Map.Entry<Integer, Double>> sortEnergy = new PriorityQueue<>((a, b) ->
b.getValue().intValue() - a.getValue().intValue());
        for (Map.Entry<Integer, Double> entry : sortMap.entrySet()) {
            sortEnergy.offer(entry);
        }

        Arrays.fill(sendOrNot, false);
        // if the energy is high, send it
        while (!sortEnergy.isEmpty() && tempData < numberOfDG * numberOfDataItemsPerDG) {
            if (sortEnergy.peek().getKey() > numberOfDG) {
                int node = sortEnergy.peek().getKey();
                tempData += numberOfStoragePerSN;
                sendOrNot[node] = true;
            }
            sortEnergy.poll();
        }

        // run MInCost
        MinCostILP minCost = new MinCostILP(dataGens, storageNodes, minCapacity,
numberOfDataItemsPerDG, numberOfStoragePerSN, treeMap, adjacencyList1, close);
```

```
        minCost.solve("original", sendOrNot);

        // energy cost and resilience level
        Cell minCostEnrgy = energyrow.createCell(8);
        minCostEnrgy.setCellValue(minCost.getILPObj());
        Cell minCostResilience = energyrow.createCell(9);
        minCostResilience.setCellValue(minCost.getResilience());

        ArrayList<Integer> mindeadNodes = minCost.getDeadNodes();
        if (mindeadNodes.size() > 0) {
          Cell MinCostDead = energyrow.createCell(12);
          MinCostDead.setCellValue(Arrays.toString(mindeadNodes.toArray()));
        }

        /*------------------------- extra information ------------------------*/
        // base energy / storage / gap
        Cell baseEnergy = energyrow.createCell(13);
        baseEnergy.setCellValue(minCapacity);
        Cell baseStorage = energyrow.createCell(14);
        baseStorage.setCellValue(numberOfStoragePerSN);
        Cell BaseGap = energyrow.createCell(15);
        BaseGap.setCellValue(gapTolerance);

        // each run add the common difference
        numberOfDataItemsPerDG += commonDifference;
      }
    } else if (runMethod == 6){
      // for a same graph, run twice
      double gapTolerance = 0.03;

      for (int i = 0 ; i < 2; i++) {
        int tempmincapa = minCapacity;

        // copy the tree so we don't effect the original one
        Map<String, Link> copyTree = new TreeMap<>();
        for (Map.Entry<String, Link> pair : treeMap.entrySet()) {
          Link link = new Link(new Edge(pair.getValue().getEdge().getTail(),
pair.getValue().getEdge().getHead(), 0),
              pair.getValue().getDistance(), pair.getValue().getRCost(), pair.getValue().getTCost(),
pair.getValue().getSCost(),
              pair.getValue().getEnergy());
          copyTree.put(pair.getKey(), link);
        }

        for (Link link : copyTree.values()) {
          sortMap.put(link.getEdge().getHead(), link.getEnergy());
        }

        while (minCapacity > 0) {
          System.out.println("Now running energy capacity (in nano-J): " + minCapacity);

          // create new row to store nre data
          energyrow = energysheet.createRow(rowcounter++);

          /* ------------------------------ QP part ---------------------------------- */
```

```
        // CPLEX_QP
        QP_cplex qp_method = new QP_cplex(dataGens, storageNodes, numberOfDataItemsPerDG,
            numberOfStoragePerSN, treeMap ,adjacencyList1 ,close);
        qp_method.solveAllowFail(0.03, "original");

        // data items sent
        Cell cell3 = energyrow.createCell(3);
        cell3.setCellValue(qp_method.getDataSend());

        Cell cell4 = energyrow.createCell(4);
        cell4.setCellValue(qp_method.getEnergyCost(true));
        Cell cell5 = energyrow.createCell(5);
        cell5.setCellValue(qp_method.getCplexObj());

        ArrayList<Integer> deadNodes = qp_method.getDeadNodes();
        if (deadNodes.size() > 0) {
          Cell CPLEXDead = energyrow.createCell(10);
          CPLEXDead.setCellValue(Arrays.toString(deadNodes.toArray()));
        }

        /* -------------------------- network base part ------------------------------ */
        // run Algorithm
        NetworkBase networkAlgo = new NetworkBase(dataGens, storageNodes, numberOfDataItemsPerDG,
numberOfStoragePerSN, treeMap);
        boolean algOk = networkAlgo.solve();

        System.out.println();
        if (algOk) {
          Cell cell6 = energyrow.createCell(6);
          cell6.setCellValue(networkAlgo.getTotalEnergyCost());
          Cell cell7 = energyrow.createCell(7);
          cell7.setCellValue(networkAlgo.getDataResilience());
        } else {
          System.out.println("have problem: network base");
          Cell cell6 = energyrow.createCell(6);
          cell6.setCellValue("X");
          Cell cell7 = energyrow.createCell(7);
          cell7.setCellValue("X");
        }

        // check dead
        PriorityQueue<Integer> deadAlgo = networkAlgo.getDeadnodes();
        if (algOk) {
          if (!deadAlgo.isEmpty()) {
            ArrayList<Integer> deadAlgoNodes = new ArrayList<>();
            while (!deadAlgo.isEmpty()) {
              deadAlgoNodes.add(deadAlgo.poll());
            }
            Cell AlgoDead = energyrow.createCell(11);
            AlgoDead.setCellValue(Arrays.toString(deadAlgoNodes.toArray()));
          }
        } else {
          Cell AlgoDead = energyrow.createCell(11);
          AlgoDead.setCellValue("X");
        }
```

```
        /* ---------------------------------- MCF ILP part ---------------------------------- */
        // decide which node to send
        boolean[] sendOrNot = new boolean[numberOfNodes + 1];
        int tempData = 0;

        // sort the energy for the mincost usage
        PriorityQueue<Map.Entry<Integer, Double>> sortEnergy = new PriorityQueue<>((a, b) ->
b.getValue().intValue() - a.getValue().intValue());
        for (Map.Entry<Integer, Double> entry : sortMap.entrySet()) {
          sortEnergy.offer(entry);
        }

        Arrays.fill(sendOrNot, false);
        // if the energy is high, send it
        while (!sortEnergy.isEmpty() && tempData < numberOfDG * numberOfDataItemsPerDG) {
          if (sortEnergy.peek().getKey() > numberOfDG) {
            int node = sortEnergy.peek().getKey();
            tempData += numberOfStoragePerSN;
            sendOrNot[node] = true;
          }
          sortEnergy.poll();
        }

        // run MInCost
        MinCostILP minCost = new MinCostILP(dataGens, storageNodes, minCapacity,
numberOfDataItemsPerDG, numberOfStoragePerSN, treeMap, adjacencyList1, close);
        boolean cplexOK = minCost.solve("original", sendOrNot);

        // energy cost and resilience level
        if (cplexOK) {
          Cell minCostEnrgy = energyrow.createCell(8);
          minCostEnrgy.setCellValue(minCost.getILPObj());
          Cell minCostResilience = energyrow.createCell(9);
          minCostResilience.setCellValue(minCost.getResilience());
        } else {
          Cell minCostEnrgy = energyrow.createCell(8);
          minCostEnrgy.setCellValue("X");
          Cell minCostResilience = energyrow.createCell(9);
          minCostResilience.setCellValue("X");
        }

        //dead node check
        ArrayList<Integer> mindeadNodes = minCost.getDeadNodes();
        if (cplexOK) {
          if (mindeadNodes.size() > 0) {
            Cell MinCostDead = energyrow.createCell(12);
            MinCostDead.setCellValue(Arrays.toString(mindeadNodes.toArray()));
          }
        } else {
          Cell MinCostDead = energyrow.createCell(12);
          MinCostDead.setCellValue("X");
        }

        // clear the sorted part (for new generation)
```

```
            sortMap.clear();
            sortEnergy.clear();

            /*------------------------- extra information ----------------------*/
            // base energy / storage / gap
            Cell baseEnergy = energyrow.createCell(13);
            baseEnergy.setCellValue(minCapacity);
            Cell baseStorage = energyrow.createCell(14);
            baseStorage.setCellValue(numberOfStoragePerSN);
            Cell BaseGap = energyrow.createCell(15);
            BaseGap.setCellValue(gapTolerance);

            for (Link link : copyTree.values()) {
              link.setEnergy(link.getEnergy() - 200 * baseUnit);
              sortMap.put(link.getEdge().getHead(), link.getEnergy() - 200 * baseUnit);
            }
            // each run, change base capacity
            minCapacity -= 200 * baseUnit;
          }

          // generate data for another storage capacity
          rowcounter++;
          if (i < 1) {
            System.out.println("Input another capacity: ");
            numberOfStoragePerSN = scan.nextInt();
          }
          minCapacity = tempmincapa;
          copyTree.clear();
          sortMap.clear();
        }
      } else if (runMethod == 7) {
        // for a same graph, run twice
        double gapTolerance = 0.03;
        for (int i = 0 ; i < 2; i++) {
          if (i == 0) {
            // create new row to store nre data
            energyrow = energysheet.createRow(rowcounter++);

            /* ---------------------------------- QP ----------------------------------------*/
            QP_cplex qp_method = new QP_cplex(dataGens, storageNodes, numberOfDataItemsPerDG,
numberOfStoragePerSN, treeMap ,adjacencyList1 ,close);
            qp_method.solve(gapTolerance, "original");

            double[] totalSent = qp_method.getDataArray();
            double totaldata = 0;
            for (double num : totalSent) {
              totaldata += num;
            }

            Cell cell3 = energyrow.createCell(3);
            cell3.setCellValue(totaldata);

            Cell cell4 = energyrow.createCell(4);
            cell4.setCellValue(qp_method.getEnergyCost(true));
            Cell cell5 = energyrow.createCell(5);
```

```
        cell5.setCellValue(qp_method.getCplexObj());

        ArrayList<Integer> deadNodes = qp_method.getDeadNodes();
        if (deadNodes.size() > 0) {
          Cell CPLEXDead = energyrow.createCell(10);
          CPLEXDead.setCellValue(Arrays.toString(deadNodes.toArray()));
        }

        /* ------------------------------ network base ------------------------------- */
        // run Algorithm
        NetworkBase networkAlgo = new NetworkBase(dataGens, storageNodes, numberOfDataItemsPerDG,
numberOfStoragePerSN, treeMap);
        boolean algOk = networkAlgo.solve();

        Cell cell6 = energyrow.createCell(6);
        cell6.setCellValue(networkAlgo.getTotalEnergyCost());
        Cell cell7 = energyrow.createCell(7);
        cell7.setCellValue(networkAlgo.getDataResilience());

        System.out.println();
        if (!algOk) {
          System.out.println("have problem: network base");
        }
        // check dead
        PriorityQueue<Integer> deadAlgo = networkAlgo.getDeadnodes();
        if (!deadAlgo.isEmpty()) {
          ArrayList<Integer> deadAlgoNodes = new ArrayList<>();
          while (!deadAlgo.isEmpty()) {
            deadAlgoNodes.add(deadAlgo.poll());
          }
          Cell AlgoDead = energyrow.createCell(11);
          AlgoDead.setCellValue(Arrays.toString(deadAlgoNodes.toArray()));
        }

        /* ---------------------------------- MCF ILP ------------------------------------ */
        boolean[] sendOrNot = new boolean[numberOfNodes + 1];
        int tempData = 0;

        // sort the energy for the mincost usage
        PriorityQueue<Map.Entry<Integer, Double>> sortEnergy = new PriorityQueue<>((a, b) ->
            b.getValue().intValue() - a.getValue().intValue());
        for (Map.Entry<Integer, Double> entry : sortMap.entrySet()) {
          sortEnergy.offer(entry);
        }

        Arrays.fill(sendOrNot, false);
        // if the energy is high, send it
        while (!sortEnergy.isEmpty() && tempData < numberOfDG * numberOfDataItemsPerDG) {
          if (sortEnergy.peek().getKey() > numberOfDG) {
            int node = sortEnergy.peek().getKey();
            tempData += numberOfStoragePerSN;
            sendOrNot[node] = true;
          }
          sortEnergy.poll();
        }
```

```
        // run MInCost
        MinCostILP minCost = new MinCostILP(dataGens, storageNodes, minCapacity,
numberOfDataItemsPerDG,
            numberOfStoragePerSN, treeMap, adjacencyList1, close);
        minCost.solve("original", sendOrNot);

        Cell minCostEnrgy = energyrow.createCell(8);
        minCostEnrgy.setCellValue(minCost.getILPObj());
        Cell minCostResilience = energyrow.createCell(9);
        minCostResilience.setCellValue(minCost.getResilience());

        ArrayList<Integer> mindeadNodes = minCost.getDeadNodes();
        if (mindeadNodes.size() > 0) {
          Cell MinCostDead = energyrow.createCell(12);
          MinCostDead.setCellValue(Arrays.toString(mindeadNodes.toArray()));
        }

        /* -------------------------  input information ------------------------------ */
        Cell baseEnergy = energyrow.createCell(13);
        baseEnergy.setCellValue(minCapacity);
        Cell baseStorage = energyrow.createCell(14);
        baseStorage.setCellValue(numberOfStoragePerSN);
        Cell BaseGap = energyrow.createCell(15);
        BaseGap.setCellValue(gapTolerance);

        rowcounter++;
      } else {

        /* ---------------------------------- QP ----------------------------------------*/
        QP_cplex qp_method = new QP_cplex(dataGens, storageNodes, numberOfDataItemsPerDG,
numberOfStoragePerSN, treeMap ,adjacencyList1 ,close);
        qp_method.solve(gapTolerance, "original");

        double[] totalSent = qp_method.getDataArray();
        double totaldata = 0;
        for (double num : totalSent) {
          totaldata += num;
        }

        Cell cell3 = energyrow.createCell(3);
        cell3.setCellValue(totaldata);

        Cell cell4 = energyrow.createCell(4);
        cell4.setCellValue(qp_method.getEnergyCost(false));
        Cell cell5 = energyrow.createCell(5);
        cell5.setCellValue(qp_method.getCplexObj());

        ArrayList<Integer> deadNodes = qp_method.getDeadNodes();
        if (deadNodes.size() > 0) {
          Cell CPLEXDead = energyrow.createCell(10);
          CPLEXDead.setCellValue(Arrays.toString(deadNodes.toArray()));
        }

        System.out.println();
```

```
        }
      }
    } else {
      System.out.println("Wrong input!");
    }

    // for .txt files data output
//    generateFiles(treeMap, adjacencyList1);

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

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

  // receive and save cost
  double getRSCost(){
      final int K = 512; // k = 512B (from paper0)
      final double E_elec = 100 * Math.pow(10,-9); // E_elec = 100nJ/bit (from paper 1)
      double Erx = 8 * E_elec * K; // Receiving energy consumption assume is same as saving
      return Erx * 1000 * baseUnit; // make it pico J now for better number visualization during calculation
  }

  // transfer cost
  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 paper 1)
      final double Epsilon_amp = 100 * Math.pow(10,-12); // Epsilon_amp = 100 pJ/bit/squared(m) (from
paper 1)
      double Etx = E_elec * K * 8 + Epsilon_amp * K * 8 * l * l; //
      return Math.round(Etx * 1000 * baseUnit * 10000) / 10000.0; // make it pico J now for better number
visualization during calculation
  }

  /**
   * Check if the graph is connected and generate the graph
   * @param width:
   * @param height:
   * @param adjList: adjacent list of each node
   */
  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.");
    } else {
      System.out.println("Graph is not fully connected.");
      biconnectcounter++;
    }

    // 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();

  }

  /**
   * recursiveDFS use to check connection
   * @param u:
   * @param connectedNode:
   * @param adjList: adjacent list
   */
  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<>(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);
      }
    }
  }

  /**
   * generate nodes
   * @param nodeCount: node's amount
   * @param width:
   * @param height:
   */
  static 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 = Math.floor(xAxis * scale) / scale;
      yAxis = Math.floor(yAxis * scale) / scale;

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

      nodes.put(i, axis);
    }
  }

  /**
   * create node's edge (link) information
   * @param nodeCount: node's amount
   * @param tr: transfer range (upper bound)
   * @param adjList: use to record adjacent node
   */
  void populateAdjacencyList(int nodeCount, int tr, Map<Integer, Set<Integer>> adjList) {
    for(int i = 1; i <= nodeCount; i++) {
      adjList.put(i, new HashSet<>());
    }

    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("(" + node2 + ", " + node1 + ")", new Link(new Edge(node2, node1, 0), distance,
getRSCost(), getTCost(distance), getRSCost(), 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("(" + node2 + ", " + node1 + ")", new Link(new Edge(node2, node1, 1), distance,
getRSCost(), getTCost(distance), getRSCost(), energy));
          } else {
                links.put("(" + node1 + ", " + node2 + ")", new Link(new Edge(node1, node2, 1), distance,
getRSCost(), getTCost(distance), getRSCost(), energy));
          }
        }
      }
    }
  }
}
```

10.2 Network Base Algorithm Implementation

```
import java.util.*;

/**
 * Project: sensor
 * Package: PACKAGE_NAME
 * File: NetworkBase
 * Author: Shang-Lin Hsu
 * Date: Nov, 2020
 * Description: this program implements Network Base Algorithm
 */
```

```
public class NetworkBase {
   private int[] dataGens;
   private int numberOfNodes;
   private int dataPerDG;
   private int spacePerSt;
   private double totalEnergyCost;
   private double dataResilience;
   private PriorityQueue<Integer> deadnodes = new PriorityQueue<>();
   private Map<String, Link> copytreeMap;

   /**
    *
    * @return totalEnergyCost
    */
   public double getTotalEnergyCost() {
      return totalEnergyCost;
   }

   /**
    *
    * @return dataResilience
    */
   public double getDataResilience() {
      return  dataResilience;
   }

   /**
    *
    * @return deadnodes list
    */
   public PriorityQueue<Integer> getDeadnodes() {
      return deadnodes;
   }

   public NetworkBase(int[] dataGens, int[] storageNodes, int dataPerDG, int spacePerSt, Map<String, Link>
copytreeMap) {
      this.dataGens = dataGens;
      this.numberOfNodes = dataGens.length + storageNodes.length;
      this.dataPerDG = dataPerDG;
      this.spacePerSt = spacePerSt;
      this.copytreeMap = new HashMap<>(copytreeMap);
   }

   public boolean solve() {
      double totalEnergy = 0.0;
      double dataresilient = 0.0;
      Map<String, Link> treeMap = new TreeMap<String, Link>(copytreeMap);
      HashSet<Integer> set = new HashSet<>();

      List<Map.Entry<String, Link>> list = new ArrayList<Map.Entry<String, Link>>(treeMap.entrySet());

      // descending order
      Collections.sort(list, (o1, o2) -> o1.getValue().compareTo(o2.getValue()));

      int[] storageList = new int[numberOfNodes + 1];
```

```
        double[] energyList = new double[numberOfNodes + 1];

        for (int i = 1; i < dataGens.length + 1; i++) {
            storageList[i] = dataPerDG;
        }

        for (Map.Entry<String, Link> pair : treeMap.entrySet()) {
            Link link = pair.getValue();
            energyList[link.getEdge().getHead()] = link.getEnergy();
        }

//      Calling Dijkastra Algorithm
        WeighedDigraph graph = new WeighedDigraph(treeMap);
        DijkstraFind finder = new DijkstraFind(graph);

        int storageCapacity = spacePerSt;

        // use sorted map here
        for (int numOfData = 0 ; numOfData < dataGens.length * dataPerDG; numOfData++) {
            for (Map.Entry<String, Link> sortedMap : list) {
                Link link = sortedMap.getValue(); // get current link
                Map<Double, ArrayList<Integer>> map = new HashMap<>(); // list of the path, key is the result of
total cost
                PriorityQueue<Double> pq = new PriorityQueue<>(); //

                if (link.getEdge().getHead() <= dataGens.length || storageList[link.getEdge().getHead()] ==
storageCapacity){
                    continue;
                } else { // case can sent
                    // if current links head > 10, can be sent
                    // go over 10 nodes to find the shortest path
                    for (int i = 1; i <= dataGens.length; i++) {
                        ArrayList<Integer> path = finder.shortestPath(i, link.getEdge().getHead(), dataGens.length);
                        double min = 0;

                        int size = path.size();
                        if (size < 2) continue; // data generator cannot send data
                        for (int j = 0; j < size; j++) {
                            if (j == 0) {
                                String str = buildString(path, j + 1, j);

                                if (treeMap.containsKey(str)) {
                                    double headCost = treeMap.get(str).getTCost() - treeMap.get(str).getRCost();
                                    min += headCost;
                                } else {
                                    System.out.println("have prob can't find" + str);
                                    System.out.println(path.toString());
                                }

                            } else if (j == size - 1) {
                                String str = buildString(path, j - 1, j);
                                if (treeMap.containsKey(str)) {

                                    double tailRcost = treeMap.get(str).getRCost();
                                    double tailScost = treeMap.get(str).getSCost();
```

```
                min += tailRcost + tailScost;
            } else {
                System.out.println("have prob can't find" + str);
                System.out.println(path.toString());
            }

        } else {
            String str = buildString(path, j, j + 1);
            if (treeMap.containsKey(str)) {
                double middleRcost = treeMap.get(str).getTCost();
                min += middleRcost;
            } else {
                System.out.println("have prob can't find" + str);
                System.out.println(path.toString());
            }
        }
    }
    // put the path to map
    map.put(min, new ArrayList<>(path));
    pq.add(min);
}

// check if the node can send data (still have storage)
while (!pq.isEmpty() && storageList[map.get(pq.peek()).get(0)] == 0) {
    pq.poll();
}
if (pq.isEmpty()) {
    System.out.println("energy is not enough to send all data out");
    System.out.println(Arrays.toString(storageList));
    System.out.println(Arrays.toString(energyList));
    return false;
}

// send the data so add to total cost
totalEnergy += pq.peek();
storageList[map.get(pq.peek()).get(0)]--; // data generator send out one data
storageList[map.get(pq.peek()).get(map.get(pq.peek()).size() - 1)]++; // storage receive one data

// if we find min node, calculate map's value here
ArrayList<Integer> target = new ArrayList<>(map.get(pq.peek()));

// this time take out the cost
int size = target.size();
for (int j = 0; j < target.size(); j++) {
    if (j == 0) {
        String str = buildString(target, j + 1, j);
        if (treeMap.containsKey(str)) {
            double headCost = treeMap.get(str).getTCost() - treeMap.get(str).getRCost();
            energyList[target.get(j)] -= headCost;
        } else {
            System.out.println("Oppps" + str);
        }

    } else if (j == size - 1) {
        String str = buildString(target, j - 1, j);
```

```
                if (treeMap.containsKey(str)) {
                    double tailRcost = treeMap.get(str).getRCost();
                    double tailScost = treeMap.get(str).getSCost();
                    energyList[target.get(j)] -= (tailRcost + tailScost);
                } else {
                    System.out.println("Opppps" + str);
                }

            } else {
                String str = buildString(target, j, j + 1);
                if (treeMap.containsKey(str)) {
                    double middleTcost = treeMap.get(str).getTCost();
                    energyList[target.get(j)] -= middleTcost;
                } else {
                    System.out.println("Opppps" + str);
                }
            }
        }
    }

    // every loop finish will break, the data will be resorted
    break;
}

// copy the node, resort the energy
TreeMap<String, Link> cloneTreeMap = new TreeMap<>(treeMap);

// clone the map
for (Map.Entry<String, Link> entry : treeMap.entrySet()) {
    if (energyList[entry.getValue().getEdge().getHead()] < entry.getValue().getTCost()) {
        int target = entry.getValue().getEdge().getHead();
        String reverse = "(" + entry.getValue().getEdge().getHead() + ", " +
entry.getValue().getEdge().getTail() + ")";
        cloneTreeMap.remove(entry.getKey());
        cloneTreeMap.remove(reverse);
        set.add(target);
    }
}

treeMap.clear();
treeMap.putAll(cloneTreeMap);

list = new ArrayList<>(treeMap.entrySet());

// descending
Collections.sort(list, (o1, o2) -> o1.getValue().compareTo(o2.getValue()));

// after sort, regenerate the graph
graph = new WeighedDigraph(treeMap);
finder = new DijkstraFind(graph);
}

for (int i : set) {
    deadnodes.offer(i);
}
```

```
        // after all calculation the resault will be dataresilient
        for (int i = 0; i < storageList.length; i++) {
            if (i < dataGens.length + 1 && storageList[i] > 0) {
                System.out.println("Cannot deliver all data");
                System.out.println(Arrays.toString(storageList));
                System.out.println(Arrays.toString(energyList));
                return false;
            } else {
                dataresilient += storageList[i] * energyList[i];
            }
        }

        // the output data will be store in this array
        totalEnergyCost = totalEnergy;
        dataResilience = dataresilient;
        System.out.println(Arrays.toString(storageList));
        System.out.println(Arrays.toString(energyList));

        return true; // calculation complete
    }

    /**
     * build the string for map's key (node 1, node 2)
     * @param path: the edge
     * @param j: start
     * @param k: end
     * @return return key for the Map
     */
    public static String buildString (ArrayList<Integer> path, int j, int k) {
        StringBuilder sb = new StringBuilder();
        sb.append("(");
        sb.append(path.get(j));
        sb.append(", ");
        sb.append(path.get(k));
        sb.append(")");
        return sb.toString();
    }
}
```

10.3 Minimum Cost ILP Implementation

```
/**
 * Project: sensor
 * Package: PACKAGE_NAME
 * File: MinCostILP
 * Author: Shang-Lin Hsu
 * Date: Nov, 2020
 * Description: this program implement Minimum Cost Base ILP
 */
public class MinCostILP {
    private int[] dataGens;
    private int[] storageNodes;
    private int dataPerDG;
```

```
    private int spacePerSt;
    private int numberOfNodes;
    private double CplexObj = 0;
    private ArrayList<Integer> deadNodes = new ArrayList<>();
    private Map<String, Link> treeMap;
    private Map<Integer, Set<Integer>> adjacencyList1;
    private HashMap<Integer, List<Integer>> tempclose;
    private Map<String, Double> cplexresult;
    private ArrayList<Double> totalenergy;

    public MinCostILP(int[] dataGens, int[] storageNodes, int minCapacity, int dataPerDG, int spacePerSt,
            Map<String, Link> treeMap, Map<Integer, Set<Integer>> adjacencyList1, HashMap<Integer,
List<Integer>> tempclose) {
        // required network inputs
        this.dataPerDG = dataPerDG;
        this.spacePerSt = spacePerSt;
        this.dataGens = dataGens; // number of data generaters
        this.storageNodes = storageNodes; // number of storages
        this.treeMap = new HashMap<>(treeMap); // the edges
        this.adjacencyList1 = new HashMap<>(adjacencyList1); // the adjacency list
        this.tempclose = new HashMap<>(tempclose); // the closest node
        this.cplexresult = new HashMap<>();
        this.totalenergy = new ArrayList<>();
        this.numberOfNodes = dataGens.length + storageNodes.length;
    }

    /**
     *
     * @return objective function
     */
    public double getILPObj() {
        return CplexObj;
    }

    /**
     *
     * @return list of dead nodes
     */
    public ArrayList<Integer> getDeadNodes() {
        return deadNodes;
    }

    /*------------------------------- Cplex column name initialization -------------------------------*/
    /**
     * this function create the name for lp / Cplex formulation's column
     * @param treeMap: get the links
     * @param dgs: data generators
     * @param sns: storage nodes
     * @return : name for each column
     * @throws IloException :
     */
    private Map<String, IloNumVar> varName (IloCplex Cp, Map<String, Link> treeMap, int[] dgs, int[] sns)
throws IloException {

        Map<String, IloNumVar> name = new TreeMap<>();
```

```
    // create column names for each index
    // source sink to data generators
    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);
    }
    // edges between two nodes
    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);
    }
    // storage nodes to storage sink
    for (int sn : sns) {
      String str = "x" + sn + "'" + (this.adjacencyList1.size() + 1);
      IloNumVar element = Cp.numVar(0, Double.MAX_VALUE, str);
      name.put(str, element);
    }

    return name;
  }

  /*-------------------------- Calculates total energy cost ------------------------------*/
  /**
   * This method calculate the cost of the resilience (minimum cost flow)
   * @return : total cost of the path
   */
  public double getResilience() {
    double result = 0;
    double[] initialEnergy = new double[numberOfNodes + 1];
    double[] storage = new double[numberOfNodes + 1];

    for (Map.Entry<String, Link> entry: treeMap.entrySet()) {
      initialEnergy[entry.getValue().getEdge().getHead()] = entry.getValue().getEnergy();
    }

    StringBuilder path = new StringBuilder();

    for (Map.Entry<String, Double> entry : cplexresult.entrySet()){
      boolean zeroflag = false;
      List<Integer> nodes = new ArrayList<>(); // stores the current two nodes
      path.append(entry.getKey() + " : " + entry.getValue() + "\n");
      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))) {
          // any number starts at 0 is not count in the calculation
          if(curname.charAt(j) == '0'){
            zeroflag = true;
            break;
          } else {
```

```
                // this loop will take out one number form the string
                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++;
                }
                nodes.add(num);
            }
        }
    }

    // conpare with the input map to retrieve the energy cost
    if (!zeroflag) {
        int from = nodes.get(0);
        int to = nodes.get(1);

        // destination is saving sink
        if (to == numberOfNodes + 1) {
            storage[from] = entry.getValue();
            to = adjacencyList1.get(from).iterator().next();
            double value = treeMap.get("(" + to + ", " + from + ")").getSCost() * entry.getValue();
            initialEnergy[from] -= value;
        } else {
            double fromvalue = (treeMap.get("(" + from + ", " + to + ")").getTCost() - treeMap.get("(" + from
+ ", " + to + ")").getRCost()) * entry.getValue();
            double tovalue = treeMap.get("(" + from + ", " + to + ")").getRCost() * entry.getValue();
            initialEnergy[from] -= fromvalue;
            initialEnergy[to] -= tovalue;
        }
    }
}

for(int i = 1; i < storage.length; i++) {
    int deadcont = 0;
    for (int j : adjacencyList1.get(i)) {
        if (initialEnergy[i] < treeMap.get("(" + i + ", " + j + ")").getRCost() &&
                initialEnergy[i] < treeMap.get("(" + i + ", " + j + ")").getTCost() - treeMap.get("(" + i + ", " + j +
")").getRCost()) {
            deadcont++;
        }
    }
    if (deadcont == adjacencyList1.get(i).size()) {
        deadNodes.add(i);
    }
    result += storage[i] * initialEnergy[i];
}

System.out.println(Arrays.toString(storage));
System.out.println(Arrays.toString(initialEnergy));

//  File EnergyFile = new File("MinCost_Data_path_" + numberOfStoragePerSN + ".txt");
//  BufferedWriter writer_energy = new BufferedWriter(new FileWriter(EnergyFile));
//  writer_energy.write(path.toString());
//  writer_energy.close();
```

```
    return result;
}

/**
 * start solving the problem
 * @param nameOfFile: the name of the output file (if needed)
 * @param snedOrNot: high energy node's list
 * @return success or not
 * @throws IloException
 */
public boolean solve(String nameOfFile, boolean[] snedOrNot) throws IloException {

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

    // construct cplex object
    IloCplex CpFirst = new IloCplex(); // for max function to find the min off load electricity
    IloCplex CpObj = new IloCplex(); // for calculating optimized path
    // initialize the name of cplex object
    Map<String, IloNumVar> nameFirst = varName(CpFirst, treeMap, dataGens, storageNodes);
    Map<String, IloNumVar> nameObj = varName(CpFirst, treeMap, dataGens, storageNodes);


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

        IloLinearNumExpr exprFirst = CpFirst.linearNumExpr();
        IloLinearNumExpr exprMincost = 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
            exprMincost.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
            exprMincost.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
            exprMincost.addTerm(-1, nameObj.get(dir));
        }

        // if is storages add the storage sink
```

```
            if (i > dataGens.length) {
                int temp = numberOfNodes + 1;
                String dir = "x" + i + "'" + temp;
                exprFirst.addTerm(-1, nameFirst.get(dir));
                // for obj
                exprMincost.addTerm(-1, nameObj.get(dir));
            }

            // add constrain
            constraintsFirst.add(CpFirst.addEq(exprFirst, 0));
            constraintsMinCost.add(CpObj.addEq(exprMincost, 0));
        }

        // adding second constrain
        for(int i: adjacencyList1.keySet()) {
            IloLinearNumExpr exprFirst = CpFirst.linearNumExpr();
            IloLinearNumExpr exprMinCost = CpObj.linearNumExpr();
            int nei = adjacencyList1.get(i).iterator().next();

            // in part
            for(int j : adjacencyList1.get(i)) {
                String dir = "x" + j + "'" + i + "'";
                exprFirst.addTerm(treeMap.get("("+ j + ", " + i +")").getRCost(), nameFirst.get(dir));
                exprMinCost.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("("+i+", "+j+")").getRCost()) * 10000) / 10000;
                exprFirst.addTerm(temp, nameFirst.get(dir));
                exprMinCost.addTerm(temp, nameObj.get(dir));
            }

            // if is storages add the storage sink
            if (i > dataGens.length) {
                int temp = numberOfNodes + 1;
                String dir = "x" + i + "'" + temp;
                exprFirst.addTerm(treeMap.get("("+ adjacencyList1.get(i).iterator().next() + ", " + i +
")").getSCost(), nameFirst.get(dir));
                exprMinCost.addTerm(treeMap.get("("+ adjacencyList1.get(i).iterator().next() + ", " + i +
")").getSCost(), nameObj.get(dir));
            }

            // add constrain
            constraintsFirst.add(CpFirst.addLe(exprFirst, treeMap.get("("+ nei +", "+ i +")").getEnergy()));
            constraintsMinCost.add(CpObj.addLe(exprMinCost,  treeMap.get("("+ nei +", "+ i +")").getEnergy()));
        }

        // 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, dataPerDG));
    } else {
        int temp = numberOfNodes + 1;
        String dir = "x" + i + "''" + temp;
        if (snedOrNot[i]) {
            exprFirst.addTerm(1, nameFirst.get(dir));
            constraintsFirst.add(CpFirst.addLe(exprFirst, spacePerSt));
        } else {
            exprFirst.addTerm(1, nameFirst.get(dir));
            constraintsFirst.add(CpFirst.addEq(exprFirst, 0));
        }
    }
}

// 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);
CpFirst.exportModel("MinCostFirst_" + nameOfFile + ".lp");

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

// initial data items to offload
int[] dataIn = new int[dataGens.length];

Arrays.fill(dataIn, dataPerDG);

// objective value
System.out.println("Objective value: " + CpFirst.getObjValue());
if (CpFirst.getObjValue() < dataGens.length * dataPerDG) {
    System.out.println("Can not distribute all data!");
    // single generator constrains start at index 98
    for (int i = 1; i <= dataGens.length; i++) {
        // case the network cannot handle all data
        if (CpFirst.getValue(nameFirst.get("x0" + i + "'")) < 99.999) {
            dataIn[i - 1] = (int) CpFirst.getValue(nameFirst.get("x0" + i + "'"));
        }
    }
```

```
            return false;
        } else {
            System.out.println("Success!");
        }

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

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

            // dataIn[i] may change if FirstObj is not solvable (DG discard data)
            if (i <= dataGens.length) {
                String dir = "x0" + i + "'";
                exprMinCost.addTerm(1, nameObj.get(dir));
                constraintsMinCost.add(CpObj.addEq(exprMinCost, dataIn[i - 1]));
            } else {
                int temp = numberOfNodes + 1;
                String dir = "x" + i + "''" + temp;
                if (snedOrNot[i]) {
                    exprMinCost.addTerm(1, nameObj.get(dir));
                    constraintsMinCost.add(CpObj.addLe(exprMinCost, spacePerSt));
                } else {
                    exprMinCost.addTerm(1, nameObj.get(dir));
                    constraintsMinCost.add(CpObj.addEq(exprMinCost, 0));
                }
            }
        }

        // original objective function
        IloLinearNumExpr minCostObj = CpObj.linearNumExpr();
        for (Link link : treeMap.values()){
            String dir = "x" + link.getEdge().getTail() + "''" + link.getEdge().getHead() + "'";
            minCostObj.addTerm(link.getTCost(), nameObj.get(dir));
        }
        for (int i = 0; i < storageNodes.length; i++) {
            int temp = numberOfNodes + 1;
            String dir = "x" + storageNodes[i] + "''" + temp;
            int n2 = adjacencyList1.get(storageNodes[i]).iterator().next();
            minCostObj.addTerm(treeMap.get("("+ n2 + ", " + storageNodes[i] +")").getSCost(),
nameObj.get(dir));
        }

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

        // export model
//      cpMinCost.exportModel("MinCostObj_" + Lpfilename + ".lp");

        // 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()) {
```

```
            cplexresult.put(entry.getKey(), CpObj.getValue(entry.getValue()));
        }
    }
    else {
        System.out.println("Model not solved");
    }

    System.out.println("Second Objective value: " + CpObj.getObjValue());
    // file for verifying transferring path
    StringBuilder dataTpath = new StringBuilder();

    dataTpath.append("Second Obj Value: "+ CpObj.getObjValue() + "\n");

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

//      File PathFile = new File("G:\\downloads\\work\\Dr. Bin\\May\\sensor\\out_files\\Lpout_" + Lpfilename
+ ".txt");
//      BufferedWriter writer = new BufferedWriter(new FileWriter(PathFile));
//      writer.write(dataTpath.toString());
//      writer.close();

    CplexObj = CpObj.getObjValue();

    // clean up memory used
    CpFirst.end();
    CpObj.end();

    return true;
  }
}
```

10.4 Quadratic Programming Implementation

```
/**
 * Project: sensor
 * Package: PACKAGE_NAME
 * File: QP_cplex
 * Author: Shang-Lin Hsu
 * Date: Nov, 2020
 * Description: this program implements Quadratic Programming solution
 */
public class QP_cplex {
  private boolean maxDRL = false;
  private int[] dataGens;
  private int[] storageNodes;
  private double[] dataSendArray;
  private int maxDataSend = 0;
  private int dataPerDG;
  private int spacePerSt;
  private int numberOfNodes;
  private double CplexObj = 0;
  private ArrayList<Integer> deadNodes = new ArrayList<>();
```

```
  private ArrayList<Double> totalenergy;
  private Map<String, Link> treeMap;
  private Map<Integer, Set<Integer>> adjacencyList1;
  private HashMap<Integer, List<Integer>> tempclose;
  private Map<String, Double> cplexresult;


  /*------------------------------- Cplex column name initialization -------------------------------*/
  /**
   * this function create the name for lp / Cplex formulation's column
   * @param treeMap: get the links
   * @param dgs: data generators
   * @param sns: storage nodes
   * @return : name for each column
   * @throws IloException :
   */
  private Map<String, IloNumVar> varName (IloCplex Cp, Map<String, Link> treeMap, int[] dgs, int[] sns)
throws IloException {

    Map<String, IloNumVar> name = new TreeMap<>();

    // create column names for each index
    // source sink to data generators
    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);
    }
    // edges between two nodes
    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);
    }
    // storage nodes to storage sink
    for (int sn : sns) {
      String str = "x" + sn + "'" + (this.adjacencyList1.size() + 1);
      IloNumVar element = Cp.numVar(0, Double.MAX_VALUE, str);
      name.put(str, element);
    }

    return name;
  }

  /**
   * Constructor
   * @param dataGens: data generator list
   * @param storageNodes: storage list
   * @param dataPerDG: data items per generator
   * @param spacePerSt: storage space per storage nodes
   * @param treeMap: edge information
   * @param adjacencyList1: adjacent information
   * @param tempclose: the closest node
   */
  public QP_cplex(int[] dataGens, int[] storageNodes, int dataPerDG, int spacePerSt,
```

```
            Map<String, Link> treeMap, Map<Integer, Set<Integer>> adjacencyList1, HashMap<Integer,
List<Integer>> tempclose) {
        // required network inputs
        this.dataGens = dataGens; // number of data generaters
        this.storageNodes = storageNodes; // number of storages
        this.treeMap = new HashMap<>(treeMap); // the edges
        this.adjacencyList1 = new HashMap<>(adjacencyList1); // the adjacency list
        this.tempclose = new HashMap<>(tempclose); // the closest node
        this.cplexresult = new HashMap<>();
        this.totalenergy = new ArrayList<>();
        this.numberOfNodes = dataGens.length + storageNodes.length;
        this.dataPerDG = dataPerDG;
        this.spacePerSt = spacePerSt;
        this.dataSendArray = new double[dataGens.length];
    }

    /**
     * observe whether less data items sending out can reach higher DRL
     * @param maxDRL: true = enable, default = disable
     */
    public void setMethod(boolean maxDRL) {
        this.maxDRL = maxDRL;
    }

    /**
     *
     * @return return the max data items can send
     */
    public int getDataSend() {
        return maxDataSend;
    }

    /**
     *
     * @return return the max data items can send
     */
    public double[] getDataArray() {
        return dataSendArray;
    }

    /**
     *
     * @return objective function
     */
    public double getCplexObj() {
        return CplexObj;
    }

    /**
     *
     * @return list of dead nodes
     */
    public ArrayList<Integer> getDeadNodes() {
        return deadNodes;
    }
```

```
/**
 *
 * @param method : true: eliminate cycles first. false: don't care about cycles (user for only objective check)
 * @return the energy cost of QP
 */
public double getEnergyCost(boolean method) {
    double result = 0;

    if (method) {
        StringBuilder path = new StringBuilder();
        //copy the map
        Map<String, Double> clonedata = new HashMap<>(cplexresult);

        // this loop we first eliminate cycles between two nodes
        for (Map.Entry<String, Double> entry : cplexresult.entrySet()) {
            boolean zeroflag = false;
            List<Integer> nodes = new ArrayList<>();
            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))) {
                    // any number starts at 0 is not count in the calculation
                    if (curname.charAt(j) == '0') {
                        zeroflag = true;
                        // source sink
                        nodes.add(0);

                        // take out next number
                        j++;
                        String temp = curname.substring(j);

                        int t = 0;
                        int num = temp.charAt(t) - '0';
                        while (t + 1 < temp.length() && Character.isDigit(temp.charAt(t + 1))) {
                            num = num * 10 + (temp.charAt(t + 1) - '0');
                            t++;
                        }

                        // add the number
                        nodes.add(num);
                        break;
                    } else {
                        // this loop will take out one number form the string
                        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++;
                        }
                        nodes.add(num);
                    }
                }
            }
```

```
    if (!zeroflag) {
        int from = nodes.get(0);
        int to = nodes.get(1);
        String original = "x" + from + "'" + to + "'";
        String reverse = "x" + to + "'" + from + "'";
        if (cplexresult.containsKey(reverse)) {
            if (entry.getValue() > 0.0 && entry.getValue() <= clonedata.get(reverse)) {
                clonedata.put(reverse, clonedata.get(reverse) - entry.getValue());
                clonedata.put(original, 0.0);
            }
        }
    } else {
        int dgLabel = nodes.get(1);
        dataSendArray[dgLabel - 1] = entry.getValue();
    }
}

cplexresult.clear();
cplexresult.putAll(clonedata);

for (Map.Entry<String, Double> entry : cplexresult.entrySet()) {
    path.append(entry.getKey() + " : " + entry.getValue()+ "\n");
    boolean zeroflag = false;
    List<Integer> nodes = new ArrayList<>();
    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))) {
            // any number starts at 0 is not count in the calculation
            if (curname.charAt(j) == '0') {
                zeroflag = true;
                break;
            } else {
                // this loop will take out one number form the string
                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++;
                }
                nodes.add(num);
            }
        }
    }

    // conpare with the input map to retrieve the energy cost
    if (!zeroflag) {
        int from = nodes.get(0);
        int to = nodes.get(1);
        // destination is saving sink
        if (to == numberOfNodes + 1) {
            // randomly choose a neighbor of from, and get the saving cost
            to = adjacencyList1.get(from).iterator().next();
```

```
                result += treeMap.get("(" + from + ", " + to + ")").getSCost() * entry.getValue();
            } else {
                result += treeMap.get("(" + from + ", " + to + ")").getTCost() * entry.getValue();
            }
        }
    }

//      File EnergyFile = new File("CPLEX_Data_path_" + numberOfDataItemsPerDG * numberOfDG + "_" +
numberOfStoragePerSN + ".txt");
//      BufferedWriter writer_energy = new BufferedWriter(new FileWriter(EnergyFile));
//      writer_energy.write(path.toString());
//      writer_energy.close();

    } else {
        for (Map.Entry<String, Double> entry : cplexresult.entrySet()) {
            boolean zeroflag = false;
            List<Integer> nodes = new ArrayList<>();
            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))) {
                    // any number starts at 0 is not count in the calculation
                    if (curname.charAt(j) == '0') {
                        zeroflag = true;
                        break;
                    } else {
                        // this loop will take out one number form the string
                        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++;
                        }
                        nodes.add(num);
                    }
                }
            }

            if (!zeroflag) {
                int from = nodes.get(0);
                int to = nodes.get(1);
                // destination is saving sink
                if (to == numberOfNodes + 1) {
                    // randomly choose a neighbor of from, and get the saving cost
                    to = adjacencyList1.get(from).iterator().next();
                    result += treeMap.get("(" + from + ", " + to + ")").getSCost() * entry.getValue();
                } else {
                    result += treeMap.get("(" + from + ", " + to + ")").getTCost() * entry.getValue();
                }
            }
        }
    }
    return result;
}
```

```
/**
 * start solving
 * @param gapTolerance
 * @param nameOfFile : file name creates for detail check
 * @return fail or success
 * @throws IloException
 */
public boolean solve(double gapTolerance, String nameOfFile) throws IloException {

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

   // construct cplex object
   IloCplex CpFirst = new IloCplex(); // for max function to find the min off load electricity
   IloCplex CpObj = new IloCplex(); // for calculating optimized path
   // initialize the name of cplex object
   Map<String, IloNumVar> nameFirst = varName(CpFirst, treeMap, dataGens, storageNodes);
   Map<String, IloNumVar> nameObj = varName(CpFirst, treeMap, dataGens, storageNodes);

   // 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) {
         int temp = numberOfNodes + 1;
         String dir = "x" + i + "'" + temp;
```

```
            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();
        int nei = adjacencyList1.get(i).iterator().next();

        // 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("("+i+", "+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) {
            int temp = numberOfNodes + 1;
            String dir = "x" + i + "'" + temp;
            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, treeMap.get("("+ nei +", "+ i +")").getEnergy()));
        constraintsObj.add(CpObj.addLe(exprObj,  treeMap.get("("+ nei +", "+ i +")").getEnergy()));
    }

    // 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, dataPerDG));
    } else {
        int temp = numberOfNodes + 1;
        String dir = "x" + i + "'" + temp;
        exprFirst.addTerm(1, nameFirst.get(dir));
        constraintsFirst.add(CpFirst.addLe(exprFirst, spacePerSt));
    }
}

// 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);

//start solving
// check if the problem is solvable
if (CpFirst.solve()) {
    System.out.println("CpFirst Success");
} else {
    System.out.println("Model not solved");
}

// initial data items to offload
int[] dataIn = new int[dataGens.length];

Arrays.fill(dataIn, dataPerDG);

// objective value
System.out.println("Objective value: " + CpFirst.getObjValue());
if (CpFirst.getObjValue() < dataGens.length * dataPerDG) {
    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 + "'")) < dataPerDG - 0.1) {

            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]);
        }
    }
}

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));
      // observe whether less data items sending out can reach heigher DRL
      if (maxDRL) {
         constraintsObj.add(CpObj.addLe(exprObj, dataPerDG));
      } else {
         constraintsObj.add(CpObj.addEq(exprObj, dataIn[i - 1]));
      }
   } else {
      int temp = numberOfNodes + 1;
      String dir = "x" + i + "'" + temp;
      exprObj.addTerm(1, nameObj.get(dir));
      constraintsObj.add(CpObj.addLe(exprObj, spacePerSt));
   }
}

// add sencond objective function
// e part
IloLinearNumExpr enode = CpObj.linearNumExpr();

// save part
ArrayList<IloNumVar> qua = new ArrayList<>();
ArrayList<Double> quaVal = new ArrayList<>();

// the add of all objs (calculate one by one)
IloNumExpr allObj = CpObj.numExpr();

for (int i : adjacencyList1.keySet()) {
   if (i > dataGens.length) {
      IloLinearNumExpr receivenode = CpObj.linearNumExpr();
      IloLinearNumExpr outnode = CpObj.linearNumExpr();
      IloNumExpr tempObjre = CpObj.numExpr();
      IloNumExpr tempObjse = CpObj.numExpr();
      //save cost
      int head = adjacencyList1.get(i).iterator().next();
      double e = treeMap.get("(" + head + ", " + i + ")").getEnergy();
      int temp = numberOfNodes + 1;
      String savingNode = "x" + i + "'" + temp;
      // e * x ? 51 -> remaining energy
      enode.addTerm(e, nameObj.get(savingNode));

      qua.add(nameObj.get(savingNode));
      quaVal.add(-1 * treeMap.get("(" + head + ", " + i + ")").getSCost());

      // form the neighbor list
      for (int j : adjacencyList1.get(i)) {
         String receiveNode = "x" + j + "'" + i + "'";
         String sendNode = "x" + i + "'" + j + "'";
```

```
            receivenode.addTerm(-1 * treeMap.get("(" + j + ", " + i + ")").getRCost(),
nameObj.get(receiveNode));
            outnode.addTerm(-1 * (treeMap.get("(" + i + ", " + j + ")").getTCost() - treeMap.get("(" + i + ", " + j
+ ")").getRCost()), nameObj.get(sendNode));
          }

          tempObjre = CpObj.prod(nameObj.get(savingNode), receivenode);
          tempObjse = CpObj.prod(nameObj.get(savingNode), outnode);

          allObj = CpObj.sum(tempObjre, allObj);
          allObj = CpObj.sum(tempObjse, allObj);
        }
      }

      for (int i = 0 ; i < quaVal.size(); i++) {
        allObj = CpObj.sum(allObj, CpObj.prod(quaVal.get(i), qua.get(i), qua.get(i)));
      }

      allObj = CpObj.sum(enode, allObj);

      // write objective function
      CpObj.addMaximize(allObj);

      // ------------- set system parameters for QP ----------------
      CpObj.setParam(IloCplex.Param.OptimalityTarget, IloCplex.OptimalityTarget.OptimalGlobal);
      if (gapTolerance != 0) {
        CpObj.setParam(IloCplex.Param.MIP.Tolerances.MIPGap, gapTolerance);
      }

      CpObj.setParam(IloCplex.Param.MIP.Strategy.Branch, 1);
      CpObj.setParam(IloCplex.Param.Simplex.Tolerances.Markowitz, 0.1);
      CpObj.setParam(IloCplex.Param.Emphasis.MIP, 0); // Emphasize feasibility over optimality
      CpObj.setParam(IloCplex.Param.MIP.Display, 4);
      // this part eliminate MIP issue
      CpObj.setParam(IloCplex.Param.Preprocessing.Linear, 0);
      CpObj.setParam(IloCplex.Param.Emphasis.Numerical, true); // Emphasizes precision in numerically
unstable or difficult problems.
      CpObj.setParam(IloCplex.Param.Read.Scale, 1); // More aggressive scaling

      CpObj.exportModel("CPLEXObj_TwoCompare" + nameOfFile + ".lp");

      // 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()) {
          cplexresult.put(entry.getKey(), CpObj.getValue(entry.getValue()));
        }
      }
      else {
        System.out.println("Model not solved");
      }

      System.out.println("Second Objective value: " + CpObj.getObjValue());
```

```
        // file for verifying transfer 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");
        }

        // create file for data verification
//      File PathFile = new File("G:\\downloads\\eclipseJava-workspace\\SensorNetwork\\Lp_output\\Lpout_"
+ nameOfFile + ".txt");
//      BufferedWriter writer = new BufferedWriter(new FileWriter(PathFile));
//      writer.write(dataTpath.toString());
//      writer.close();

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

        // calculate the fake cost or remove cost
        DeadNodeCompute deadcomputeFake = new DeadNodeCompute(dataGens, storageNodes);
        tempp = deadcomputeFake.calculate(treeMap, tempclose, CpObj, nameObj);

        deadNodes = deadcomputeFake.getDeadNodes();
        totalenergy.addAll(tempp);

        CplexObj = CpObj.getObjValue();
        maxDataSend = (int) CpFirst.getObjValue();

        // clean up memory used
        CpFirst.end();
        CpObj.end();

        return true;
    }

    /**
     * start solving (allow calculation fail)
     * @param gapTolerance
     * @param nameOfFile file name creates for detail information
     * @return can solve or not
     */
    public boolean solveAllowFail(double gapTolerance, String nameOfFile) {
        try {
            List<IloRange> constraintsFirst = new ArrayList<IloRange>();
            List<IloRange> constraintsObj = new ArrayList<IloRange>();

            // construct cplex object
            IloCplex CpFirst = new IloCplex(); // for max function to find the min off load electricity
            IloCplex CpObj = new IloCplex(); // for calculating optimized path
            // initialize the name of cplex object
            Map<String, IloNumVar> nameFirst = varName(CpFirst, treeMap, dataGens, storageNodes);
            Map<String, IloNumVar> nameObj = varName(CpFirst, treeMap, dataGens, storageNodes);

            // 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) {
        int temp = numberOfNodes + 1;
        String dir = "x" + i + "'" + temp;
        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();
    int nei = adjacencyList1.get(i).iterator().next();

    // 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("(" + i + ", " + 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) {
                int temp = numberOfNodes + 1;
                String dir = "x" + i + "'" + temp;
                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, treeMap.get("(" + nei + ", " + i + ")").getEnergy()));
            constraintsObj.add(CpObj.addLe(exprObj, treeMap.get("(" + nei + ", " + i + ")").getEnergy()));
        }

        // 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, dataPerDG));
            } else {
                int temp = numberOfNodes + 1;
                String dir = "x" + i + "'" + temp;
                exprFirst.addTerm(1, nameFirst.get(dir));
                constraintsFirst.add(CpFirst.addLe(exprFirst, spacePerSt));
            }
        }

        // 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);
```

```
//start solving
// check if the problem is solvable
if (CpFirst.solve()) {
   System.out.println("CpFirst Success");
} else {
   System.out.println("Model not solved");
}

// initial data items to offload
int[] dataIn = new int[dataGens.length];

Arrays.fill(dataIn, dataPerDG);

// objective value
System.out.println("Objective value: " + CpFirst.getObjValue());
if (CpFirst.getObjValue() < dataGens.length * dataPerDG) {
   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 + "'")) < dataPerDG - 0.1) {

         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]);
      }
   }
}

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));
      // observe whether less data items sending out can reach heigher DRL
      if (maxDRL) {
         constraintsObj.add(CpObj.addLe(exprObj, dataPerDG));
      } else {
         constraintsObj.add(CpObj.addEq(exprObj, dataIn[i - 1]));
      }
   } else {
      int temp = numberOfNodes + 1;
      String dir = "x" + i + "'" + temp;
      exprObj.addTerm(1, nameObj.get(dir));
      constraintsObj.add(CpObj.addLe(exprObj, spacePerSt));
   }
}
```

```
        // add sencond objective function
        // e part
        IloLinearNumExpr enode = CpObj.linearNumExpr();

        // save part
        ArrayList<IloNumVar> qua = new ArrayList<>();
        ArrayList<Double> quaVal = new ArrayList<>();

        // the add of all objs (calculate one by one)
        IloNumExpr allObj = CpObj.numExpr();

        for (int i : adjacencyList1.keySet()) {
           if (i > dataGens.length) {
              IloLinearNumExpr receivenode = CpObj.linearNumExpr();
              IloLinearNumExpr outnode = CpObj.linearNumExpr();
              IloNumExpr tempObjre = CpObj.numExpr();
              IloNumExpr tempObjse = CpObj.numExpr();
              //save cost
              int head = adjacencyList1.get(i).iterator().next();
              double e = treeMap.get("(" + head + ", " + i + ")").getEnergy();
              int temp = numberOfNodes + 1;
              String savingNode = "x" + i + "'" + temp;
              // e * x ? 51 -> remaining energy
              enode.addTerm(e, nameObj.get(savingNode));

              qua.add(nameObj.get(savingNode));
              quaVal.add(-1 * treeMap.get("(" + head + ", " + i + ")").getSCost());

              // form the neighbor list
              for (int j : adjacencyList1.get(i)) {
                 String receiveNode = "x" + j + "'" + i + "'";
                 String sendNode = "x" + i + "'" + j + "'";

                 receivenode.addTerm(-1 * treeMap.get("(" + j + ", " + i + ")").getRCost(),
nameObj.get(receiveNode));
                 outnode.addTerm(-1 * (treeMap.get("(" + i + ", " + j + ")").getTCost() - treeMap.get("(" + i + ", "
+ j + ")").getRCost()), nameObj.get(sendNode));
              }

              tempObjre = CpObj.prod(nameObj.get(savingNode), receivenode);
              tempObjse = CpObj.prod(nameObj.get(savingNode), outnode);

              allObj = CpObj.sum(tempObjre, allObj);
              allObj = CpObj.sum(tempObjse, allObj);
           }
        }

        for (int i = 0; i < quaVal.size(); i++) {
           allObj = CpObj.sum(allObj, CpObj.prod(quaVal.get(i), qua.get(i), qua.get(i)));
        }

        allObj = CpObj.sum(enode, allObj);

        // write objective function
```

```
        CpObj.addMaximize(allObj);

        // ------------ set system parameters for QP ----------------
        CpObj.setParam(IloCplex.Param.OptimalityTarget, IloCplex.OptimalityTarget.OptimalGlobal);
        if (gapTolerance != 0) {
           CpObj.setParam(IloCplex.Param.MIP.Tolerances.MIPGap, gapTolerance);
        }

        CpObj.setParam(IloCplex.Param.MIP.Strategy.Branch, -1);
        CpObj.setParam(IloCplex.Param.Simplex.Tolerances.Markowitz, 0.1);
        CpObj.setParam(IloCplex.Param.Emphasis.MIP, 0); // Emphasize feasibility over optimality
        CpObj.setParam(IloCplex.Param.MIP.Display, 4);
        // this part eliminate MIP issue
        CpObj.setParam(IloCplex.Param.Preprocessing.Linear, 0);
        CpObj.setParam(IloCplex.Param.Emphasis.Numerical, true); // Emphasizes precision in numerically
unstable or difficult problems.
        CpObj.setParam(IloCplex.Param.Read.Scale, 1); // More aggressive scaling

        CpObj.exportModel("CPLEXObj_TwoCompare" + nameOfFile + ".lp");

        // 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()) {
              cplexresult.put(entry.getKey(), CpObj.getValue(entry.getValue()));
           }
        } else {
           System.out.println("Model not solved");
        }

        System.out.println("Second Objective value: " + CpObj.getObjValue());
        // file for verifying transfer 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");
        }

        // create file for data verification
//      File PathFile = new File("G:\\downloads\\eclipseJava-workspace\\SensorNetwork\\Lp_output\\Lpout_"
+ nameOfFile + ".txt");
//      BufferedWriter writer = new BufferedWriter(new FileWriter(PathFile));
//      writer.write(dataTpath.toString());
//      writer.close();

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

        // calculate the fake cost or remove cost
        DeadNodeCompute deadcomputeFake = new DeadNodeCompute(dataGens, storageNodes);
        tempp = deadcomputeFake.calculate(treeMap, tempclose, CpObj, nameObj);
```

```
            deadNodes = deadcomputeFake.getDeadNodes();
            totalenergy.addAll(tempp);

            CplexObj = CpObj.getObjValue();
            maxDataSend = (int) CpFirst.getObjValue();

            // clean up memory used
            CpFirst.end();
            CpObj.end();

            return true;
        } catch (IloException e) {
            System.out.println("QP Error");
            return false;
        }
    }
}
```

10.5 Quadratic Dead Nodes Computation

```
/**
 * Project: sensor
 * Package: PACKAGE_NAME
 * File: DeadNodeCompute
 * Author: Shang-Lin Hsu
 * Date: Nov, 2020
 * Description: This program is used to calculate the dead nodes' information and energy cost
 */
public class DeadNodeCompute {

    private int[] dataGens;
    private int[] storageNodes;
    private int numberOfNodes;
    private HashMap<Integer, Double> totaldataitems;
    private ArrayList<Integer> deadNodes;

    /**
     *
     * @param dataGens: data generator list
     * @param storageNodes: stoge node's list
     */
    public DeadNodeCompute(int[] dataGens, int[] storageNodes) {
        this.deadNodes = new ArrayList<>();
        this.dataGens = dataGens;
        this.storageNodes = storageNodes;
        this.numberOfNodes = dataGens.length + storageNodes.length;
        this.totaldataitems = new HashMap<>();
    }

    /**
     *
     * @return dead node's list
     */
```

```
  public ArrayList<Integer> getDeadNodes() {
    return deadNodes;
  }

  /**
   *
   * @param treeMap: edge information
   * @param tempclose: closest node
   * @param CpIn: cplex variable data
   * @param cpresult: result of the data transmission path
   * @return each node's cost
   * @throws IloException
   */
  public ArrayList<Double> calculate(Map<String, Link> treeMap, HashMap<Integer, List<Integer>>
tempclose, IloCplex CpIn,
                     Map<String, IloNumVar> cpresult) throws IloException {

    // this map contains cost for each node
    ArrayList<Double> back = new ArrayList<>();
    HashMap<Integer,List<Double>> map = new HashMap<>();

    // 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<>();
    List<Double> tempres = new ArrayList<>(); // temp is for numbers at a line from input file

    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);
        }

        // calculate target node's energy cost
        for (List<Double> re : res) {
            int transNode = (int) Math.floor(re.get(0));
            int disNode = (int) Math.floor(re.get(1));
            double items = re.get(2);

            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;

                map.get(transNode).set(0, map.get(transNode).get(0) + totalTcost); // tansferNode's Tcost
                map.get(disNode).set(1, map.get(disNode).get(1) + totalRcost); // receiveNode's Rcost

            } else {
                // case save data
                map.get(transNode).set(2, nodeScost.getOrDefault(transNode, 0.0) * items);

            }
        }

        // for text output
        // 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 + storageNodes.length];
        for (int i = 0; i < combine.length; i++) {
            if (i < dataGens.length) {
```

```
            combine[i] = dataGens[i];
        } else {
            combine[i] = storageNodes[i - dataGens.length];
        }
    }

    // calculate total cost (0 + 1 + 2)
    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));
        // System.out.println("Node "+ i + ": "+ map.get(i).get(0) + " " + map.get(i).get(1) + " " + map.get(i).get(2)
+ " " + map.get(i).get(3));
        // add the energy cost result to send back
        back.add(map.get(i).get(3));
        // calculate weather the node is dead
        if (i <= dataGens.length) { // source nodes
            //  current energy   +   energy cost to rely (transfer + receive) data to closest node  >   the minCapacity
user identified
            if (map.get(i).get(3) + treeMap.get("(" + i + ", " + tempclose.get(i).get(0) + ")").getTCost() +
                    treeMap.get("(" + i + ", " + tempclose.get(i).get(0) + ")").getRCost() >= treeMap.get("(" +
tempclose.get(i).get(0) + ", " + i + ")").getEnergy()) {
                energy_mincostoutput.append(", status: DEAD!").append(";\r\n");
                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 (map.get(i).get(3) + treeMap.get("(" + tempclose.get(i).get(0) + ", " + i + ")").getSCost() >=
treeMap.get("(" + tempclose.get(i).get(0) + ", " + i + ")").getEnergy() ||
                    map.get(i).get(3) + treeMap.get("(" + i + ", " + tempclose.get(i).get(0) + ")").getTCost() +
                            treeMap.get("(" + i + ", " + tempclose.get(i).get(0) + ")").getRCost() >= treeMap.get("(" +
tempclose.get(i).get(0) + ", " + i + ")").getEnergy()) {
                energy_mincostoutput.append(", status: DEAD!").append(";\r\n");
                deadNodes.add(i);
            } else {
                energy_mincostoutput.append(", status: Good").append(";\r\n");
            }
        }
    }
    totaldataitems.clear();
    return back;
  }
}
```

10.6 Network 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 + ")";
    }
}
```

10.7 Network Link

```
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;
    }
  }
}
```

10.8 Network 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;
  }

}
```

10.9 Network Connection Check – Dijkstra

```
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();
  }

  /**
   * 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<>();
    weight = new HashMap<>();
    pq = new PriorityQueue<>(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;
    }
  };
```

```
}
```

10.10 Dijkstra Weighed Diagram

```
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());
    currentEdges.add(newEdge);
```

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

10.11 Dijkstra Weighed Diagram Edges

```
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; }
}
```

10.12 Sensor Network Graph Produce

```
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(0, 10, 100); // 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(0, 110, 255); // 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);
      }
    }

    //Label the nodes
    g2.setColor(Color.red);
    for (int i = 0; i < graphPoints.size(); i++) {
      int x = graphPoints.get(i).x - ovalSize / 2;
      int y = graphPoints.get(i).y - ovalSize / 2;
      g2.setFont(new Font("TimesRoman", Font.PLAIN, 18));
      g2.drawString(""+(i+1), x, y);
    }
  }
```

```
  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;
  }
}
```