REVISITING NETWORK FLOWS: A SENSOR NETWORK PERSPECTIVE

A Thesis by

FNU Nilofar

Bachelor of Electronics & Communications, VTU, India, 2005

Submitted to the Department of Electrical Engineering and Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

December 2012

© Copyright 2012 by FNU Nilofar

All Rights Reserved

REVISITING NETWORK FLOWS: A SENSOR NETWORK PERSPECTIVE

The following faculty members have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science, with a major in Computer Networking.

____

Bin Tang, Committee Chair

____

Rajiv Bagai, Committee Member

____

Bayram Yildirim, Committee Member

DEDICATION

To my father

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to my advisor, Dr. Bin Tang for his excellent guidance, continuous support and remarkable patience all throughout my research. This thesis would not have been possible without his expertise help and his valuable suggestions that helped me in completing it successfully.

I would like to extend my immense gratitude to members of my committee, Dr. Rajiv Bagai and Dr. Bayram Yildirim for offering their helpful suggestions and valuable time on my thesis.

I am indebted to my family who believed in me and supported me morally all throughout my education. I would also like to thank my friends for being there for me always.

ABSTRACT

In wireless sensor networks, data preservation has become a key-challenging problem. Data generated by some sensor nodes is huge and due to limited storage space in a sensor node, the data generating nodes have to offload the data to nodes with available storage space and high battery power. The data needs to be preserved in these nodes until the base station collect it. In this thesis, data preservation problem in sensor networks is modeled as network flow problems and it is solved by using network flow algorithms while considering the specific sensor network parameters such as battery power and storage capacity of sensor nodes. The load-balancing data preservation algorithm maximizes the minimum energy left among the nodes that store data and minimizes the total cost for data redistribution. We also formulated the data-preserving problem, with limited battery power in each node and minimized the total energy consumption of data preservation. In addition, we also studied and analyzed the feasibility of data preservation when each node has limited battery power.

TABLE OF CONTENTS

Chapter Page

1. INTRODUCTION 1

1.1. Contribution of Thesis 3

2. LITERATURE REVIEW 4

2.1. Background and Motivation 4

2.1.1. Wireless Sensor Networks 4

2.1.2. Challenges 5

2.2. Related Work 5

2.3. Organization of Thesis 7

3. LOAD-BALANCING DATA PRESERVATION ALGORITHM 8

3.1. Network Model and Problem Formulation 8

3.1.1. Formulating Data Redistribution Problem 9

3.1.2. Formulating Data Preservation Problem 10

3.2. Load-Balancing Algorithm 12

3.2.1. Algorithm to Find Multiple Shortest Path between Two Nodes 12

3.2.2. Load-Balancing Data Preservation Algorithm 14

3.3. Performance Evaluation 16

4. MINIMUM ENERGY DATA PRESERVATION 19

4.1. Introduction 19

4.2. Proposed Network Model and Problem Formulation 23

TABLE OF CONTENTS (continued)

Chapter Page

4.2.1. Proposed Network Model....23

4.2.2. Proposed Energy Model....23

4.2.3. Problem Formulation....24

4.3. Performance Evaluation of Results....25

4.3.1. Energy Levels....25

4.3.2. Data Items....27

4.3.3. Data Generators....30

5. CONCLUSION AND FUTURE WORK....33

5.1. Conclusion....33

5.2. Future Work....33

REFERENCES....35

APPENDICES....38

A. Load-balancing Data Preservation Programs....39

B. Program to Generate Network for Minimum Energy Data Preservation....61

REVISITING NETWORK FLOWS: A SENSOR NETWORK PERSPECTIVE

CHAPTER 1

INTRODUCTION

A wireless sensor network (WSN) is a wireless network of small computing devices that sense and store physical or environmental parameters such as temperature, light, pressure, sound [1] and smell [2]. Some of the WSN applications such as visual and acoustic sensor networks and multimedia surveillance networks generate huge amount of data in a specific area of the sensor network. The sensor nodes in that specific area that generate huge data are called Data Generators (DG). Due to limited storage capacity in a sensor, data generators can quickly exhaust their storage capacity and need to redistribute the data to neighboring nodes in the network to prevent any data loss. We study how to redistribute such huge amount of data to fully utilize the storage capacity of the sensor network and at the same time minimize the total energy consumption during the whole data redistribution process. We call this process data preservation in sensor networks.

In this thesis, we model data preservation in sensor networks as network flow problems and solve it using network flow algorithms while considering the specific sensor network parameters such as battery power and storage capacity of sensor nodes. In basic terms network flow is to move an entity (electricity, a consumer product, a vehicle, a packet) from one node to another in an underlying network, under different physical constraint of the networks such as edge capacities and edge costs. Network flow is a problem domain that is at the cusp between several fields of enquiry, including applied mathematics, computer science, engineering, management and operation research. There

has been a set of foundation from which we get the key ideas of network flow theory and established networks (graphs) as useful mathematical objects for representing many physical systems [3]. Network flow addresses key issues like, shortest path problem, maximum flow problem and minimum cost flow problem. Utilizing some of the algorithms of network flow and graph theory, the problems in a sensor network can be resolved.

As important is utilizing the storage capacity of the network efficiently while redistributing the data, it is equally significant to preserve the battery power of the network to prolong the network lifetime. Performing data preservation without enough battery power in the sensor nodes could lead to delay and loss of data, which in turn could cause network failure [4]. Thus considering battery power or energy as an individual parameter of every sensor node during data redistribution becomes very critical. Based on this approach the thesis is divided into two stages as explained below.

In the first stage we study the data redistribution problem in sensor networks and propose load-balancing algorithm that performs minimum cost data redistribution while maximizing the minimum energy left among destination nodes using minimum-cost flow algorithm. Here destination nodes are the nodes to which the data from the data generators is redistributed. In the second stage we aim to minimize the total energy consumption of data preservation considering each node has limited amount of battery power and study and analyze the feasibility of data preservation. We use maximum flow algorithm to check the feasibility and give a low cost solution in terms of energy consumption using minimum cost flow algorithm. We also compare the performance from both maximum flow and minimum cost flow algorithms in terms of the energy spent to redistribute the given amount of data. In both the stages, the goal is towards making the network more energy

aware while also concentrating on fully utilizing the available storage capacity of the network for data offloading.

1.1 Contribution of Thesis

The thesis is divided into two stages as follows:

- In the first stage, we study the data preservation problem in data intensive and intermittently connected sensor networks.
- We propose load-balancing algorithm, which not only minimizes the redistribution cost but also maximizes the minimum energy left among destination nodes after the redistribution in the network.
- In the second stage study the minimum cost data preservation when energy levels are low at each individual sensor node in the network. And analyze the performance of maximum flow and minimum cost flow algorithms, at various energy levels for different network scenarios.

CHAPTER 2

LITERATURE REVIEW

2.1 Background and Motivation

2.1.1 Wireless Sensor Networks

Wireless Sensor Networks (WSN) is a wireless network of small computing devices with limited resources such as scarce energy power and limited storage capacity. The core functionality of these devices is to sense physical or environmental conditions around them, such as temperature, pressure, air-quality, sound, vibrations etc and process it and then either store it for later retrieval or forward the data to sink nodes. With the emergence of modern sensory sources such as microphones, cameras, RFID readers and telescopes, much advanced industrial and commercial applications of WSN have been developed.

Some commercial applications of WSN are traffic control and management, precision agriculture, health monitoring, intelligent home applications, electricity substation monitoring, road monitoring, flood detection, fire detection and some industrial applications of WSN are factory automation, process control, real-time monitoring of machinery health and detection of liquid/gas leakage, real-time inventory management and monitoring of contaminated areas, etc. The data generated by some of the sensor nodes at a specific area in these applications is massive. Such sensor nodes are called as data generators. And due to the very nature of sensor node having limited storage capacity, data generators can quickly exhaust their storage capacity to accommodate all the data generated by them. To prevent any data loss, data generators need to offload their data to

other sensor nodes in the network with any available storage. This process is called data-redistribution. And the process of data collection by the sink nodes from the intermediate nodes or data generators is called data retrieval.

2.1.2 Challenges

Maximizing the utilization of storage capacity while minimizing the total energy consumption during data redistribution process has become one of the key challenges in the world of WSN. Utilizing the storage efficiently, while keeping the energy consumption to minimum is the key for superior performance of any sensor network. We study how to utilize storage ably and energy economically, such that the lifetime of network is amplified using some of the network flow algorithms.

Minimizing the energy consumption for data redistribution in sensor networks has been the main focus of many researches. It is observed that following a minimum cost path for redistributing the data would in turn lead to minimizing the energy conservation of the network. While it is true, however energy of each individual sensor node has to be taken into consideration, to understand the effect of energy levels in the process of data redistribution.

2.2 Related Work

Energy conserving problem has been the main focus of several studies in the field of sensor network. Minimum-cost flow and maximum flow algorithms have been used extensively in various contexts, such as, to minimize the data redistribution cost or to maximize the usage of storage capacity in the network or to maximize the sensor network lifetime. Rohini et al. [5] proposes to minimize the total energy consumption of data

redistribution process by formulating the data redistribution problem as minimum cost flow problem. We also use this approach to find the minimum cost solution to redistribute the data, but the difference between [5] and this thesis is, the shortest path used to redistribute the data from a DG to a destination node is chosen randomly in [5], but in our paper, we utilize the idea of multiple shortest paths that exist between any two nodes and choose the shortest path strategically, that is based on the energy levels of intermediate nodes that lay in a path between a DG and a destination node, hence maximizing the network lifetime. And also in [5], they consider the energy levels to be high enough in all the nodes, to achieve minimum cost for redistribution, but in the second stage of this paper, we show that even with low energy levels minimum cost redistribution can be feasible.

Xiang et al. [6] study the data preservation problem in intermittently connected sensor networks. They study the energy depletion induced data loss and provide maximum-flow based algorithm to maximize data preservation time in the network by maximizing the minimum energy left among the destination nodes after redistribution. We are also trying to achieve this objective but the main difference between [6] and this paper is we also minimize the total energy consumption while maximizing the minimum energy left among destination nodes. Our approach is based on minimum cost flow algorithm hence achieving minimum cost for redistribution while also maximizing the data preservation time.

Patel et al. [7] also study about minimizing the cost of sending data packets from sensor nodes to base stations. They propose Minimum-cost Capacity-constrained Routing protocol (MCCR) based on minimum cost flow algorithm that claims to minimize the

energy consumed in routing while being within the capacity limits of each sensor node and wireless links and Maximum Lifetime Capacity-constrained Routing (MLCR) protocol that maximizes the time until the first battery drains its energy. We also propose minimum cost flow algorithm based solution for data offloading but the difference between [7] and this thesis is we consider both energy and capacity of each sensor node and we maximize the data preservation time by maximizing the minimum energy left among destination nodes and we achieve both these goals in our load-balancing data preservation algorithm.

2.3 Organization of Thesis

The rest of the thesis is organized as follows. In chapter 3, we study data preservation problem and propose load-balancing data preservation algorithm. In chapter 4 we study the feasibility of minimum cost with low energy levels at a sensor node and then compare the results from maximum flow and minimum cost flow algorithms. Finally in Chapter 5 we make conclusion and present scope for future work.

CHAPTER 3

LOAD-BALANCING DATA PRESERVATION

3.1 Network Model and Problem Formulation

In our network model, there are some sensor nodes generating large amount of data, with total size much larger than their storage capacities. Such sensor nodes are referred as data generators. The data generated is modeled as sequence of raw data items, each data item with unit size. Every sensor node has limited storage capacity and can only hold finite number of data items. Data generators are the sensor nodes that collect more data than their actual storage capacity and hence need to offload some of their data to other sensor node that have free space. The objective of data redistribution problem is to offload the data items from the data generators to other sensor nodes to fully utilize the storage capacity of the sensor network, while minimizing the energy consumption of the sensor network and also load-balancing the individual energy consumption at different sensor nodes.

We represent our sensor network as grid network with $G (V, E)$ where $V = \{1, 2, ..., N\}$ is the set of N nodes and E is set of edges. Every node i has storage capacity c_i and energy e_i. Let d_{ij} be the cost in terms of shortest path distance (number of hops) between nodes i and j. Let $V_s = \{1, 2, ..., p\}$ be the set of data generators and p as total number of data generators. Let t be total number of data items to be redistributed in the network and let $D = \{D_1, D_2, ..., D_p\}$ be set of data items corresponding to p data generators. Let $DG(i) \in V_s$ denote the data generator of data item D_i. Let P_i be the distribution path of data item D_i, say $P_i : DG(i), x(i), ...,$

r(i), where *r(i)* is the destination node of the data item D_i and *x(i)* is the intermediate node between *DG(i)* and *r(i)*.

Each data generator redistributes one data item at a time. For our energy cost model, we use the number of hops to measure the energy consumption of redistributing the data item. The redistribution cost for *DG(i)* with D_i number of data items to redistribute, is the sum of number of hops to redistribute all D_i data items. The total redistribution cost of the sensor network is defined as sum of the redistribution cost of all the data generators. The goal here is to redistribute the data items from the data generators into the network with minimum redistribution cost. *Without loss of generality we assume that the total size of the data items to be redistributed is less than or equal to the total available storage space in the network.*

3.1.1. Formulating Data Redistribution Problem

To formulate our problem, let *D* be set of data items to be redistributed in the whole network and let *DG(i)* be the data item *i*'s data generator, where $i \in D$. A redistribution function is defined as *r : D -> V*, indicating data item $i \in D$ is redistributed to node $r(i) \in V$ via the shortest path between *DG(i)* and *r(i)*. Our goal is to find such a redistribution function r to minimize the total redistribution cost:

$$\sum_{i \in D} d_{DG(i)r(i)},$$

under the constraint that number of data items redistributed to node *j* is not more than node *j*'s available storage capacity, i.e.,

$$|\{i \mid i \in D, r(i) = j\}| \le c_j, \qquad \text{for all } j \in V$$

Let E'_i denote node *i*'s energy level after the distribution of all data items is done, and let x_{ij} be the energy cost incurred by node *i* in the process of relaying data item D_j from node *j* to *r(j)*, and let E'_i denote node *i*'s energy level after all *t* data items are redistributed. Then,

$$E'_i = E_i - \sum_{j=1}^{t} x_{ij}, \qquad \forall i \in V$$

where $x_{ij} = 1$ if $i \in P_j - \{DG(j), r(j)\}$, and $x_{ij} = 0.5$ if $i \in \{DG(j), r(j)\}$ and $x_{ij} = 0$ otherwise. Here, *i* could be the data generator or destination node of D_j (with energy cost of 0.5) or an intermediate node of D_j (with energy cost one), or not involved with the distribution at all (with energy cost zero).

3.1.2 Formulating Data Preservation Problem

The objective of data preservation problem is to find a distribution function r and set of $\mathcal{P} = \{P_1, P_2, \ldots, P_t\}$, to redistribute each of the *t* data items, such that the minimum energy among all the destination nodes is maximized post redistribution, i.e.,

$$\max_{r,P} \min_{1 \le i \le t} E'_{r(i)},$$

under the energy constraint that

$$E'_i \ge 0, \qquad \forall i \in V,$$

implying that any sensor node cannot spend more energy than its initial energy level.

We transform the grid network into a bipartite graph with every data generator having an edge to all non-DG nodes with the edge cost being number of hops from the DG to sensor node, as shown in Figure 1.

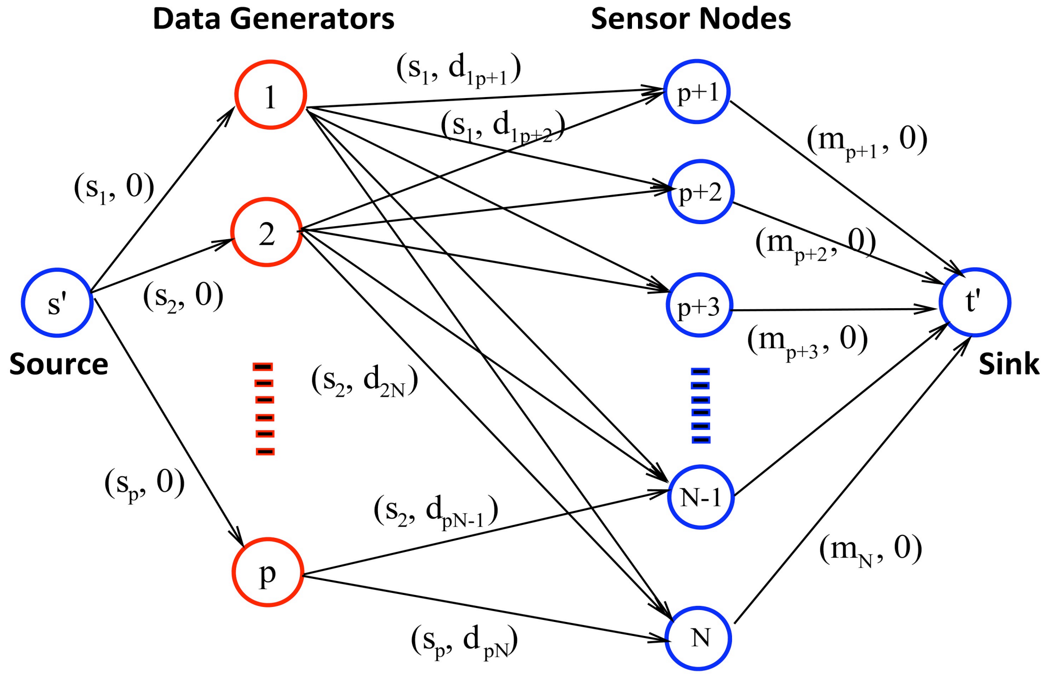

Fig. 1. Data redistribution problem is equal to minimum cost flow problem [6].

Figure 1, is given as input to the minimum cost flow algorithm [5]. The output from minimum cost program is the cost to redistribute the data items from one or multiple DGs to the destination nodes. Along with the cost it also outputs the destination nodes chosen for redistribution for every data item of every data generator. Now, based on this output of the destination nodes used for redistribution, we introduce load-balancing algorithm, which uses the same destination nodes for redistribution but it utilizes the idea of multiple shortest paths and strategically chooses the shortest path between them.

3.2 Load-balancing Data Preservation Algorithm

The objective of this algorithm is to maximize the minimum energy left among the destination/intermediate nodes, while also achieving minimum cost to redistribute the data. Before explaining this algorithm, we need to first understand another algorithm that finds multiple shortest paths between any given two nodes in a grid network.

3.2.1 Algorithm to find Multiple Shortest Paths between Two Nodes

This algorithm finds multiple shortest paths that exist between any two nodes in a grid network based on x and y coordinates of a node.

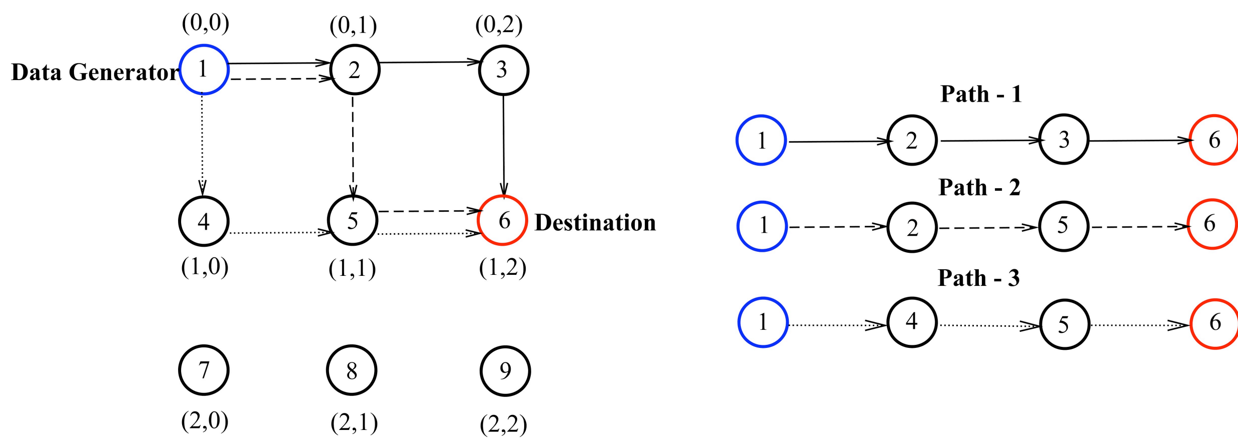

Fig. 2. In a 3x3 grid network, node 1 is data generator and node 6 is destination node. There are 3 shortest paths between node 1 and node 6, as shown above.

```
BEGIN
1.   findShortestPaths(sx,sy,dx,dy)
2.        sx =0, sy=0, dx=1, dy=2, x=sx, y=sy;
3.        vertDiff = sy-dy, horiDiff = sx-dx;
4.        vt= vertDiff, hr= horiDiff, l=0;
5.        max = (vertDiff > 0 ? vertDiff: - vertDiff)+( horiDiff > 0 ? horiDiff: - horiDiff);
6.        getPath (x, y, vertDiff, horiDiff, vt, hr, l, max)
7.                  if (vt == 0  && hr == 0) { RETURN addNode(x,y); }
8.                  ym = vertDiff > 0 ?  -1 : 1;
9.                  xm = horiDiff > 0 ?  -1 : 1;
10.                 if (vt!= 0){a = getPath (y+ym, x, vertDiff, horiDiff, vt+ym, hr, l+1, max)};
11.                 if (hr!=0) {b = getPath (y, x+xm, vertDiff, horiDiff, vt, hr+xm, l+1, max)};
12.                 for(Path p: a) { addNode(x,y)};
13.                 for(Path p : b){ addNode(x,y)};
14.                 RETURN (a+b);
15.       end getPath;
16.   end findShortestPaths;
END
```

Let *(0,0)* be the *x* and *y* co-ordinates of data generator node 1, say *(x1, y1)* and *(1,2)* be *x* and *y* coordinates of destination node 6, say *(x2,y2)*. If *(x1-x2)* is the horizontal difference of *x* coordinates of these two nodes and *(y1-y2)* is the vertical difference of y coordinates, then max is the absolute sum of these vertical and horizontal differences i.e. *|(x1-x2)|+|(y1-y2)|*. The value of max will then denote the shortest distance (in terms of number of hops) between these two nodes, in this example, value of max is 3 i.e. |(0-1)|+|(0-2)| meaning, the shortest distance between node 1 and 6 is 3 hops. It then uses a recursive function (from line 6 to line 15 below) to find all the paths between the source

and destination with the intermediate nodes, such that the hop distance in the paths found does not exceed by max. Consider example shown in Figure 2, the recursive function finds all the paths between node 1 and node 6 with distance of 3. As shown in lines 8 and 9, ym and xm have the value 1 or -1 based on the vertical and horizontal differences. The value of ym is then added to y and vt (y co-ordinate and vertical difference) unless the value of vt is not zero, for the horizontal recursion, similarly, value of xm is added to x and hr (x co-ordinate and horizontal difference) unless the value of hr is not zero for the vertical recursion. These vertical and horizontal recursions keep adding nodes to their respective paths until it reaches destination at line 7. Once the function reaches the destination, it then returns all the nodes added in both directions recursively. The value of level is incremented every time getPath() function is called recursively, making sure that the nodes added to the path does not exceed the max value.

3.2.2 Load-balancing Data Preservation Algorithm

If the path between the DG and a destination node has more than one intermediate node, the energy of these intermediate relaying nodes becomes very critical to preserve. Because if these intermediate nodes do not have enough energy to relay the data items, there is a possibility of network failure. To prevent this kind of situation, we propose the load-balancing algorithm, which makes sure that intermediate nodes chosen have enough energy before they are used as relay nodes because it mainly chooses the intermediate nodes in such a way that minimum energy among these nodes is maximized, preserving the data in the network for longer time.

If DG_i represent the data generators and d_i the data items the DG_i has to offload, then let r_i be the corresponding destination node. And the path between DG_i -> r_i has zero or more intermediate nodes.

Algorithm:

BEGIN

1. for each DG_i,
2. for each of data item of DG_i: d_i
3. Get the destination node r_i from minimum cost flow program;
4. Get the x an y coordinates of DG_i (sx,sy) and r_i (dx,dy);
5. Get *k* shortest paths, say *P[k]* from *findShortestPaths(sx,sy,dx,dy) (section 3.2.1);*
6. for each path $p_j \in P[k]$
7. Get *MinimumEnergy[j]* = node n_i with minimum energy in path p_j;
8. end for;
9. Find *MaximumMinimumEnergy = max(MinimumEnergy[k]);*
10. Get the corresponding path, p_j, with *MaximumMinimumEnergy* ;
11. RETURN p_j to redistribute the data item d_i;
12. end for;
13. end for;

END.

As explained in 3.2 we use the output from the minimum cost flow algorithm [5] in the load-balancing algorithm to find the destination node of every data item (line 3). Once it has the destination node, it finds out all the shortest paths available between the DG and the destination node (line 5) using the algorithm explained in 3.3.1. Then it follows a three-step process, to choose the best path among all the shortest paths. In the first step, it gets

the node with minimum energy in each shortest path from all the paths (line, 6,7,8). In the second step at line 9, it gets the node with highest-minimum energy out of all the nodes found in first step. In the third step at line 10, it chooses the path, which has the node with maximum-minimum-energy and returns it. Then the data item is distributed along this shortest path. This process is repeated for redistributing every data item. This way the program ensures that nodes with least amount of energy are not used up, hence maximizing the minimum energy left among destination nodes, solving the data preservation problem. Since we use minimum cost program to find out the destination nodes, the node balancing also achieves the minimum cost for data redistribution.

3.3 Performance Evaluation of Load-balancing Algorithm

Figures 3, 4 and 5 show a comparison of minimum energy left among the destination nodes between the minimum cost and load-balancing programs. For a fair comparison of results, in case of minimum cost program, we choose random shortest path for redistribution and use the same initial energy array for both the programs. We compare the results in various scenarios for a 10x10 grid network for different number of data items. Every node has initial storage capacity of 10 and energy of 1000. Here we assume energy level is high enough for redistribution. In Figure 3, we choose one DG in the center of the grid network and perform the redistribution of different number of data items. As the number of data items increase, the minimum energy left among the destination nodes in higher in case of load-balancing algorithm. In Figure 4, we do the same comparison with two DGs in the center of the network, for different number of data items and again the minimum energy left among the destination nodes is again higher in case of load-balancing

algorithm. In Figure 5, the comparison is shown for two DGs far apart in the grid network and the results are again better for load-balancing algorithm.

Fig 3. Minimum Energy Left among the destination nodes in 10x10 network with one DG at 55 with initial storage capacity as 10 and initial energy as 1000 at each sensor node.

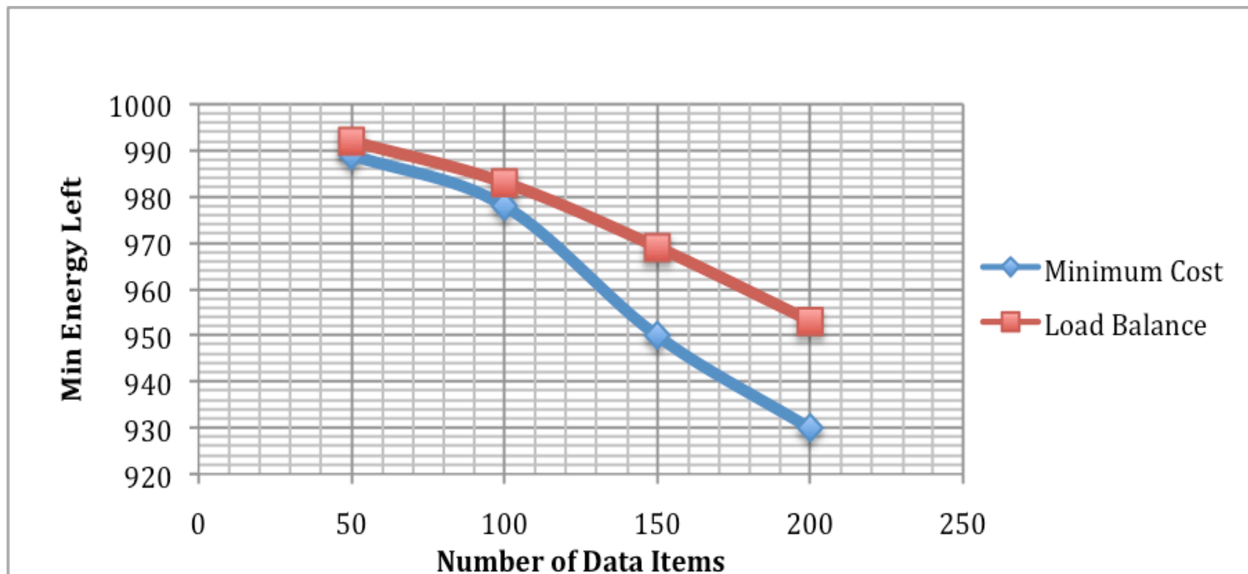

Fig 4. Minimum Energy Left among the destination nodes in 10x10 network with two DGs near by at 55 and 56 with initial storage capacity as 10 and initial energy as 1000 at each sensor node.

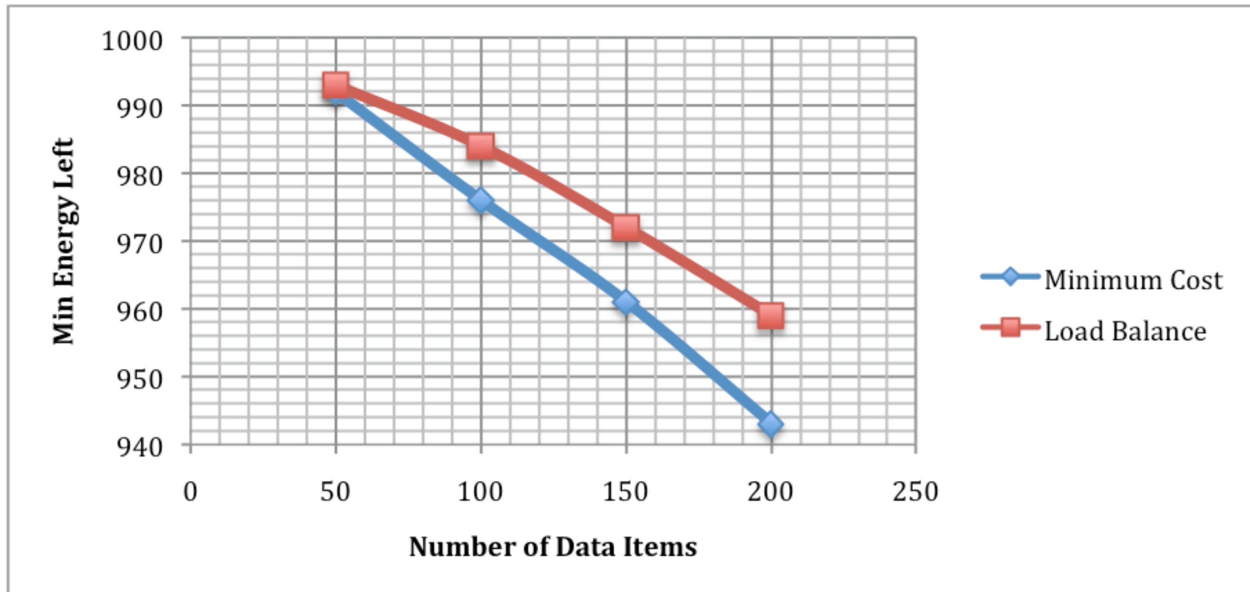

Fig 5. Minimum Energy Left among the destination nodes in 10x10 network with two DGs far apart at 12 and 89 with initial storage capacity of 10 and initial energy of 1000 at each sensor node.

CHAPTER 4

MINIMUM ENERGY DATA PRESERVATION

4.1 Introduction

In Chapter 3, we proposed load-balancing data preservation algorithm that had one of its objective to achieve minimum cost for data redistribution. We accomplished minimum cost for redistribution with the assumption that sensor nodes have high energy levels before redistribution. In this chapter we study the data redistribution problem with low energy levels at individual sensor nodes and show that minimum cost solution is still feasible. Lets take an example shown in Figure 6.1, a simple linear network with 4 nodes and say node 1 and 3 as data generators and node 2 and 4 as destination nodes. Lets assume node 1 has 20 data items to offload and say each node has initial energy level of 100. To offload 20 data items from node 1 to node 4, intermediate nodes (nodes 2 and 3) need 20 unit of energy (0.5 for receiving and 0.5 for sending for every data item). With 100 units of energy at every sensor node, this redistribution was easily feasible. But now if the initial energy level is reduced to say 10 units at every sensor node, then this redistribution becomes unfeasible because the intermediate nodes do not have enough energy to relay 20 data items. In this section we show that with the energy level of 10, minimum cost redistribution is feasible for up to 10 data items of the data generator at node 1.

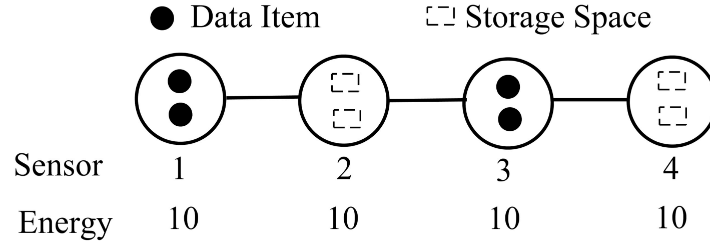

Fig. 6.1. A simple linear network with nodes 1 & 3 as data generators, each with 2 data items to offload and nodes 2 & 4 as destination nodes each with 2 units of storage space.

We transform Figure 6.1 into split node graph as shown in Figure 6.2. Figure 6.1 shows a simple linear network consisting of 4 nodes. Node 1 and 3 are the data generators and nodes 2 and 4 are regular sensor nodes. Each of node i has initial energy and capacity e_i and c_i respectively. Nodes 1 and 2 have d_1 and d_2 data items to offload respectively. Figure 6.2 show the transformed split-node graph:

- Every node is split into two; node i is split into node i' as in-node and node i'' as out-node. All the incoming edges to the node i is now to node i' and all the outgoing edges from node i are now from node i''.
- Source and sink nodes are added. Source node is connected to the in-node of data generator with edge capacity as data items, node i has to redistribute. And all the non-DG out-nodes are connected to sink nodes, with edge capacity as the storage capacity, c_i of non-DG node i.
- The edge capacity between split nodes of DG_i is (e_i + $d_i/2$) i.e. the energy level of DG_i plus half the data items the DG_i needs to redistribute.

- The edge capacity between split nodes of regular node *i* is (e_i + $c_i/2$) i.e. energy level at node *i* + half the storage capacity of node *i*, as shown in the Figure 6.2.

- The edge cost between the source and data generators is zero and the edge cost between the split nodes is zero and the edge cost between the neighbors is 1, indicating the one hop distance between them. The edge cost between the non-DG nodes and sink node is also zero.

The addition of edge capacity by half the number of data items between split DG nodes and by half the storage capacity between split non-DG nodes is because we need to accommodate proposed energy model in the transformed graph. In the 0.5 energy model, we assume that every sensor node spends 0.5 unit of energy to send or receive one data item. Thus a data generator need to spend $d/2$ units of energy to send d data items. But in the minimum cost flow algorithm, to send a unit of data over the split edges, it reduces one unit of energy instead of 0.5 and to send d data items, it spends extra d/2 amount of energy, which is not accurate according to our energy model. So we are adding same extra d/2 amount of energy to the initial energy level. And similarly non-DG nodes actually need to spend $c/2$ units of energy to receive, c data items, where c is the storage capacity of the sensor node. But in the algorithm it reduces c amount of energy to receive c data items, once gain to balance-out the extra energy spent by the algorithm, we add c/2 amount of energy to initial energy level of non-DG nodes. The edge capacity between neighboring nodes is infinity because we do not want to use finite values that will limit the data flow other than energy and storage capacity of the sensor nodes.

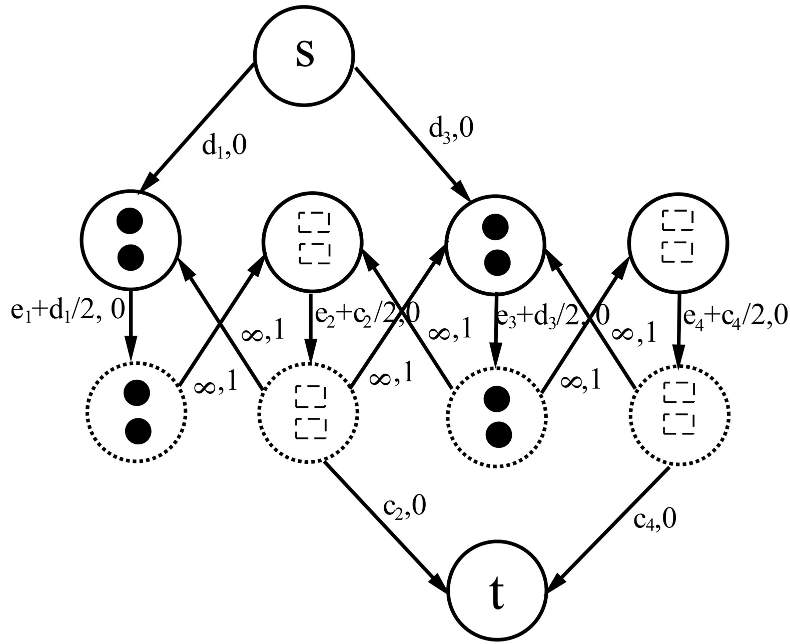

Fig. 6.2. Transformed graph of the linear network shown in Fig. 6.1.

This data preservation problem is equal to minimum cost flow problem because the goal here is to minimize the total energy consumption while being within the energy limits of every sensor node involved in data redistribution process. The edge capacity between the split nodes is the energy of a sensor node, so running the minimum cost flow algorithm makes sure that the energy required to offload, relay and receive a data item will be within the energy limits of the node involved in redistribution and the result will be minimum cost for data redistribution but in terms of energy. Thus, minimizing the total energy consumption overall in the network eventually maximizes the data preserving time in the network.

4.2 Proposed Network Model and Problem Formulation

4.2.1 Network Model

The sensor network is represented as an undirected connected graph, $G\ (V,E)$, where $V=\{1,2, ..., N\}$ is the set of N nodes, and E is set of edges. If two nodes are in the communication range of each other, they have an edge between them. Let p be the number of data generators and let V_s denote set of p data generators, $V_s=\{1,2, ..., p\}$. Let q be the total number of data items to be distributed in the network. Let c_i be the storage capacity available at sensor node i, where $i \in V$. If $i \in V_s$ then $c_i = 0$, implying that data generators have no free space available. If $i \in V - V_s$ then, $c_i \geq 0$, implying that a non-DG node i can accommodate c_i data items. And also we assume that total number of data items is less than or equal the total storage available in the network i.e. $q \leq \sum_{i=1}^{N} c_i$.

4.2.2. Energy Model

We assume that if a sensor node sends a unit of data item, it would cost 0.5 unit of energy and if it receives a data item it would cost 0.5 unit of energy. Therefore, when a DG offloads one data item, it would cost 0.5 unit of energy and similarly when a destination node receives a data item, it would cost 0.5 unit of energy and if a node serves as relay node, it costs one unit of energy i.e. 0.5 for receiving and 0.5 for sending. So the total energy required to offload one data item from a DG to a destination node would cost (0.5+no.of intermediate nodes+0.5). For example in Figure 6.1, the total energy required to offload one data item from node 1 to node 4 would be equal to 0.5(at node 1)+1(at node 2)+1(at node 3)+0.5(at node 4) = 3, which is equal to number of hops between node 1 and node 4.

Therefore, we can say that energy consumption is equal to number of hops the data item traverses from DG to destination node. Energy required for data retrieval from sensor nodes is beyond the scope of this thesis.

4.2.3. Problem Formulation:

Let d_{ij} be the cost in terms of shortest path distance (number of hops) between nodes i and j. To formulate our problem, let D denote the set of data items to be redistributed in the network and let $DG(i)$ is the data item i's data generator, where $i \in D$. A redistribution function is defined as $r: D \to V$, indicating data item $i \in D$ is redistributed to node $r(i) \in V$ via the shortest path between $DG(i)$ and $r(i)$. Our goal is to find such a redistribution function r to minimize the total redistribution cost:

$$\min \sum_{i \in D} d_{DG(i)r(i)},$$

under the constraint that the number of data items redistributed to node j is not more than node j's available storage capacity, i.e.,

$$|\{i \mid i \in D, r(i) = j\}| \leq c_j, \qquad \text{for all } j \in V.$$

and number of data items to be redistributed is less than or equal to total network storage capacity, i.e.,

$$q \leq \sum_{i=1}^{N} c_i$$

The network model shown in Figure 6.2 basically limits the data flow based on both energy and capacity of the sensor nodes. When this network model is used in conjunction

with maximum flow algorithm, it outputs the maximum data items to offload without exceeding the energy and storage capacity limits in the network. And when this flow value is used with minimum cost flow algorithm with the same network model, we would minimize the total energy consumption with low energy levels in the network.

4.3 Performance Evaluation of results from Maximum Flow and Minimum Cost Flow algorithm.

We consider several network scenarios and compare the results from maximum flow and minimum cost flow algorithms. The value being compared here is total energy consumed to distribute the given data items in various cases.

4.3.1. Energy Levels

To distribute the data items from a DG to destination nodes, intermediate nodes need to have enough energy in order to relay the data items. In this section we analyze how varying the energy levels of the sensor nodes affect the flow value in the network for different network sizes and analyze which algorithm performs better.

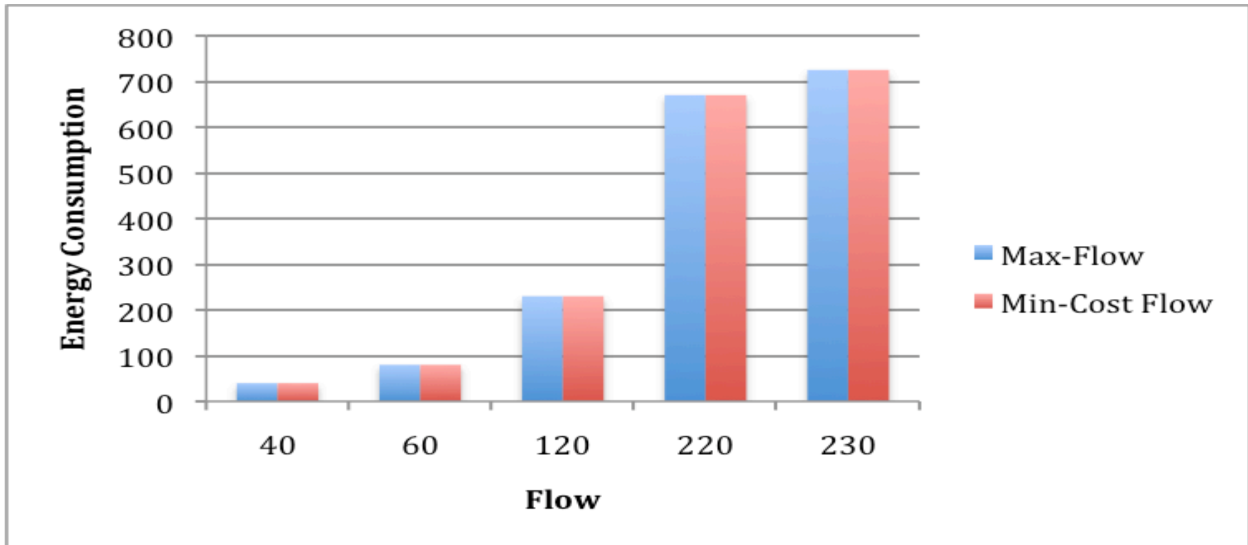

Fig. 7. Energy Consumption vs Flow for a 5x5 network for various energy levels from 10 to 75.

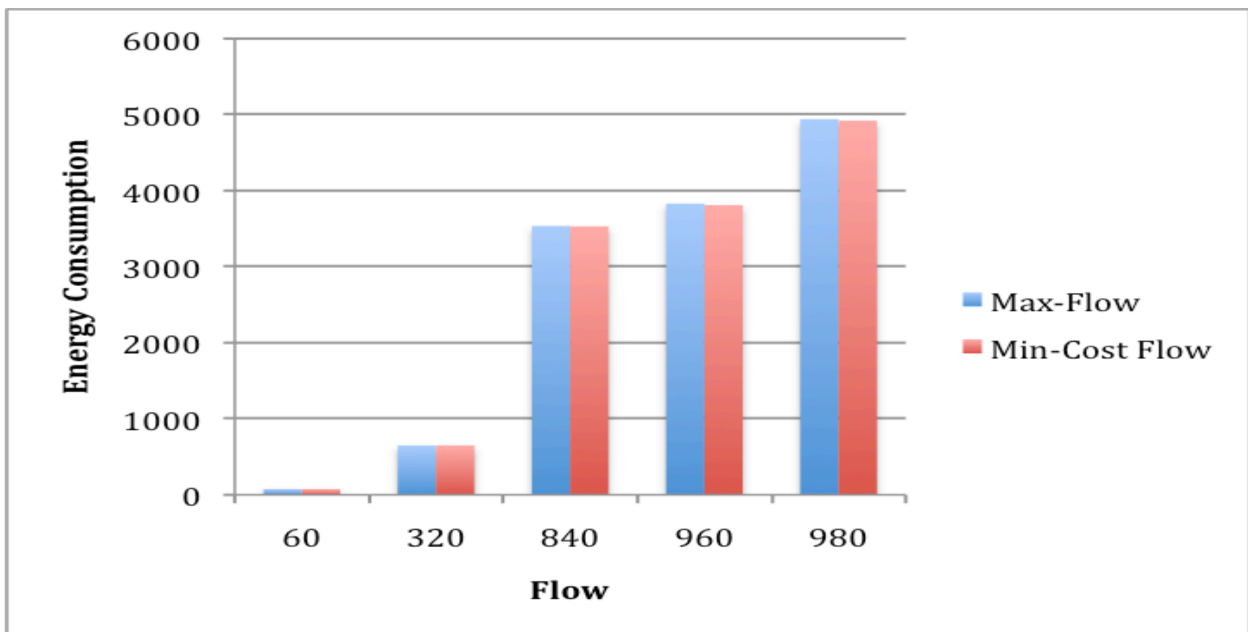

Fig. 8. Energy Consumption vs Flow for a 10x10 network for varying energy levels from 10 to 160.

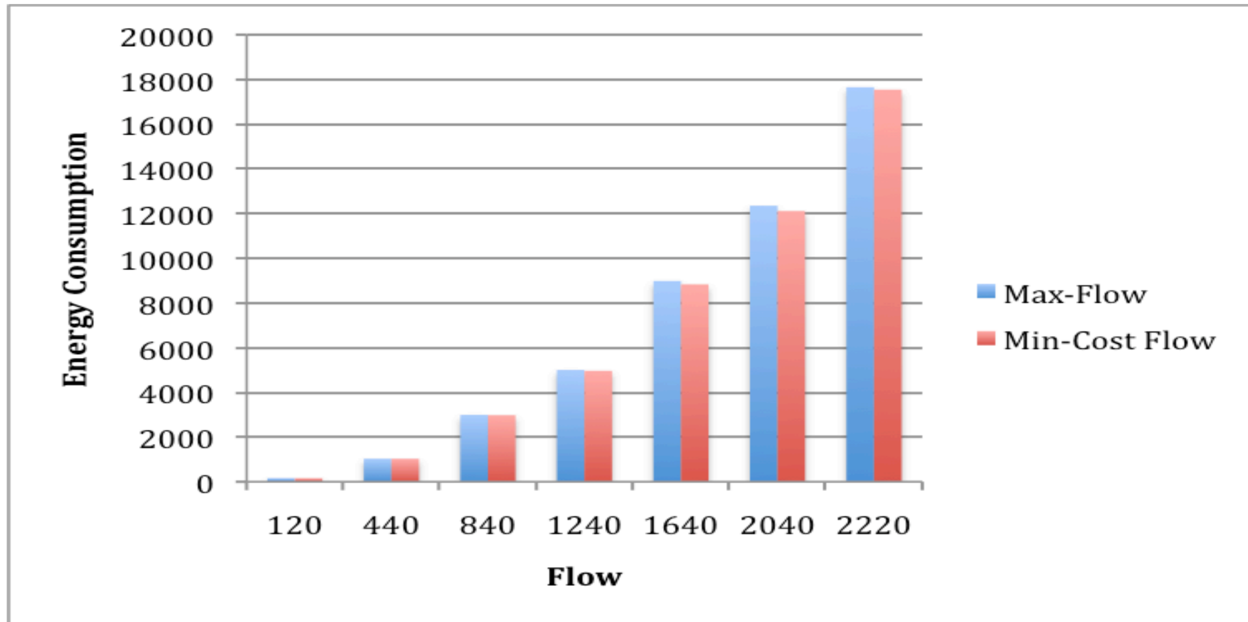

Fig. 9. Energy Consumption vs Flow for a 15x15 network for varying energy levels from 10 to 300.

It is clear from the above Figures 7, 8 and 9 that as energy level is increased, flow value increases. It is also very interesting to note that for network size 5*5, there was no difference in the energy consumption from both the algorithms, however when the network size and flow value increases, the energy consumption from minimum cost flow algorithm is lesser than the energy consumption by maximum flow algorithm.

4.3.2. Data Items

In this case, energy level and capacity are constant i.e. energy level is 50 for 5x5 network, 160 for 10x10 network and 300 for 15x15 network and capacity as 10 in all the sensor nodes. Then we change the number of data items to be offloaded, keeping the number of data generators constant and record the energy consumption from both the algorithms. One of the observations is, as the number of data items is increased, the energy

consumption increases in all the three network scenarios. But the energy consumption from maximum flow algorithm is more when compared to the energy consumption from minimum cost flow algorithm as we increase the data items and the network size, as shown below in Figure 11 and Figure 12.

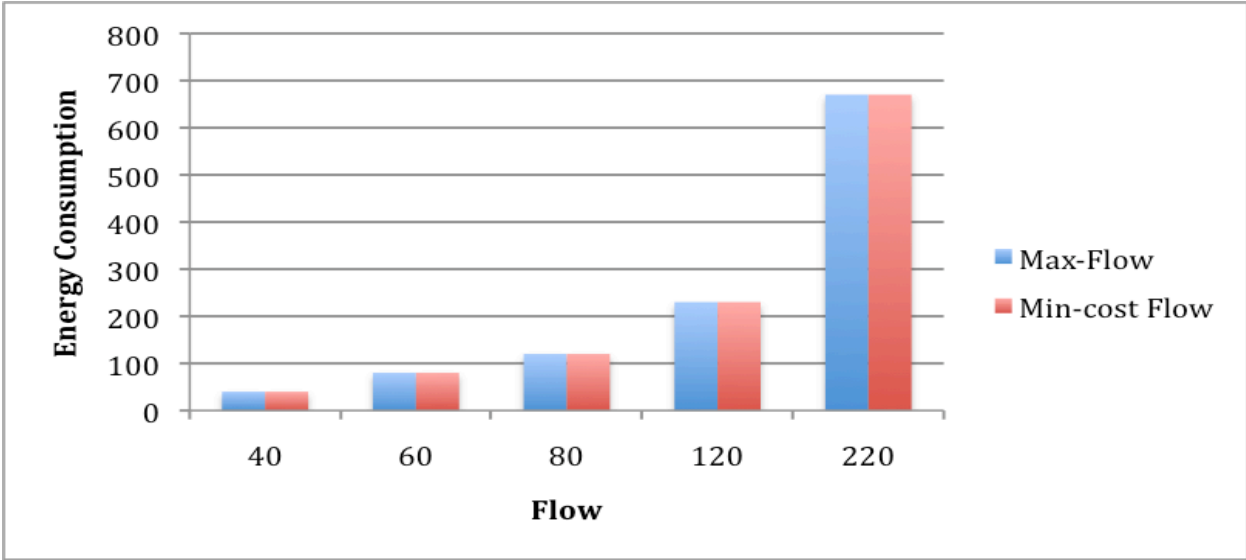

Fig. 10. Energy consumption vs Flow for a 5x5 network for variable data items from 40 to 220.

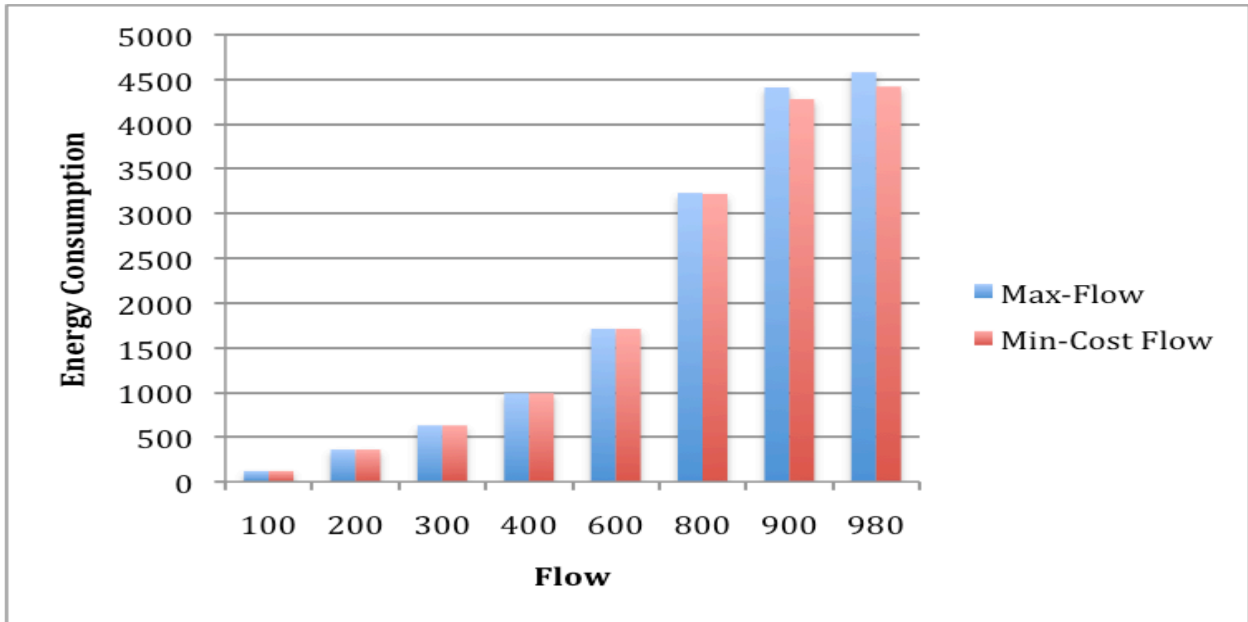

Fig. 11. Energy consumption vs Flow for a 10x10 network for variable data items from 100 to 980.

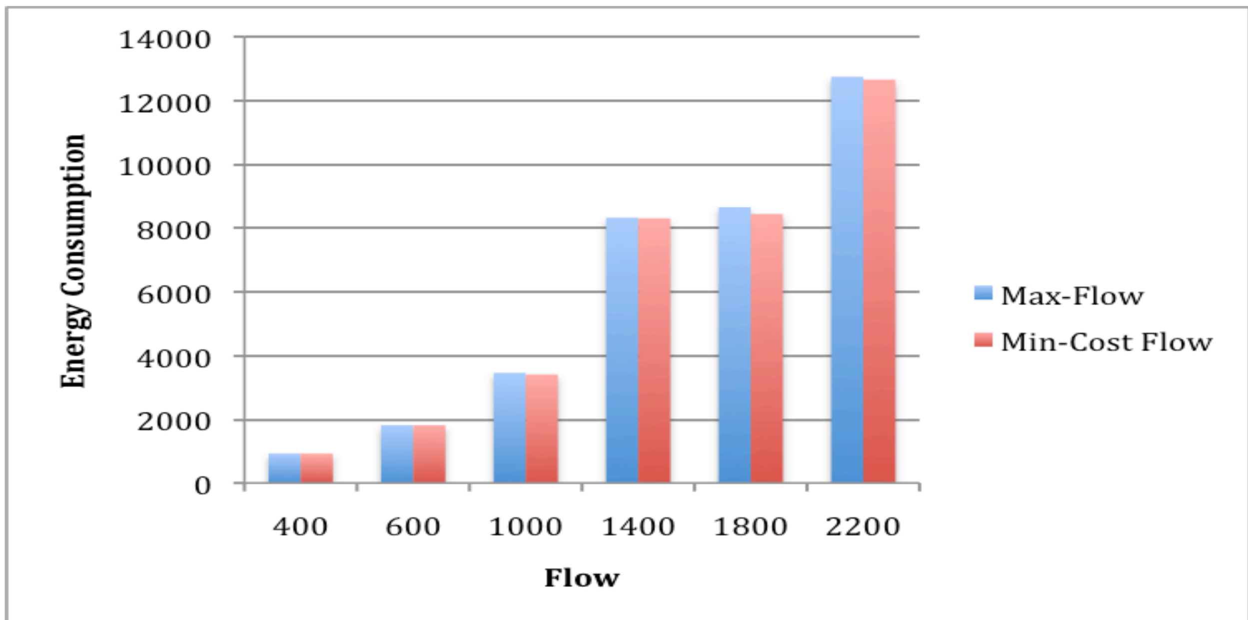

Fig. 12. Energy consumption vs Flow for a 15x15 network for variable data items from 400 to 2200.

Overall we can say that as the network size is increased and as the number of data items are increased, minimum cost flow algorithm performs much better than maximum flow algorithm.

4.3.3. Data Generators

In this case we keep the energy, capacity and data items to a constant value and change the number of data generators. Here we try to analyze if the number of data generators make any difference in the energy consumption. Energy levels for 5x5 network is 50, for 10x10 network it is 160 and 300 for 15x15 network at every sensor node. Capacity is 10 at every sensor node for all the network sizes.

For 5x5 network there is no difference in the energy consumption from maximum flow and minimum cost flow algorithms. But for 10x10 and 15x15 network, the energy consumption from maximum flow is higher then minimum cost flow program for all the cases.

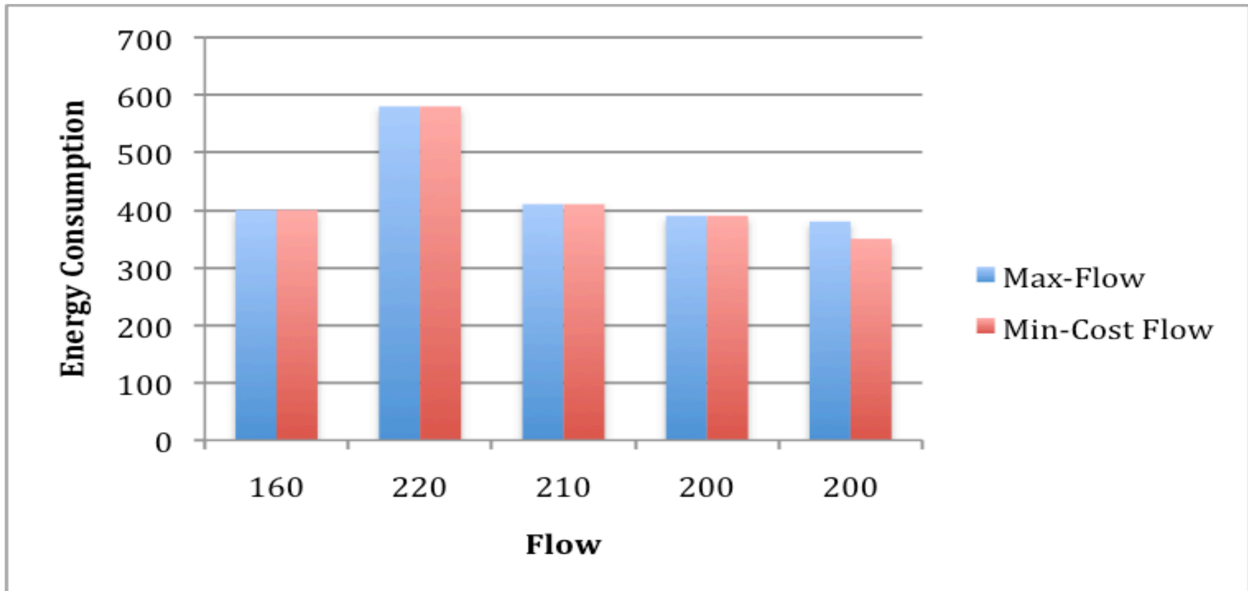

Fig. 13. Energy consumption vs Flow for a 5x5 network for varying data generators from 1 to 5.

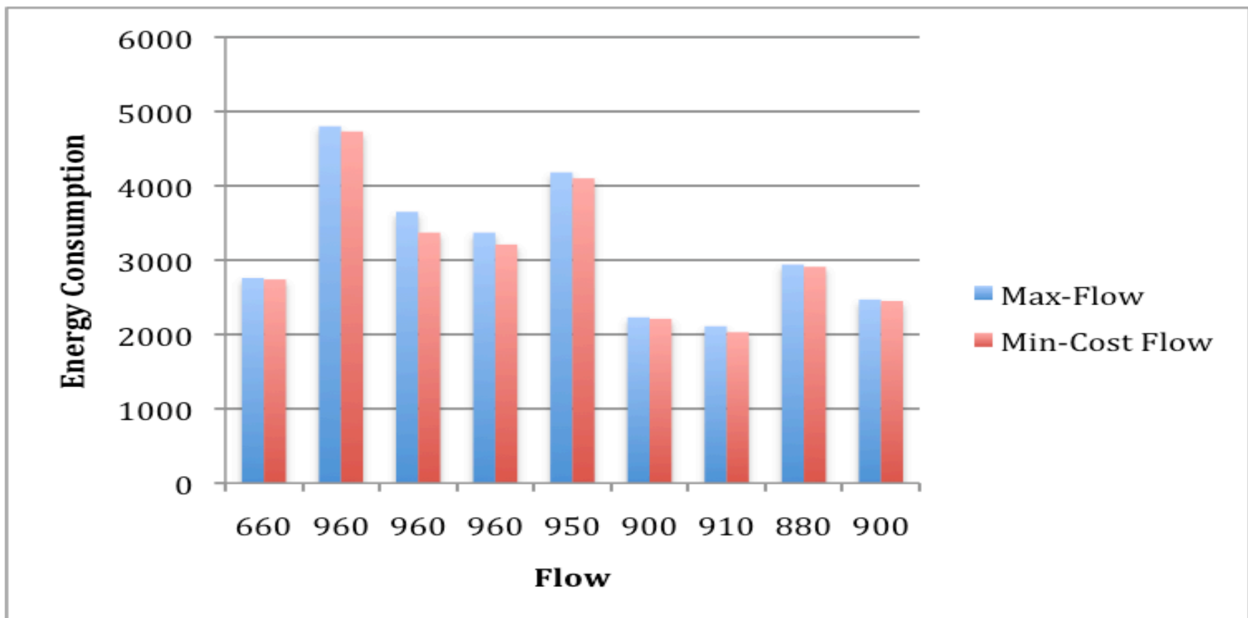

Fig. 14. Energy consumption vs Flow for a 10x10 network for varying data generators from 1 to 9.

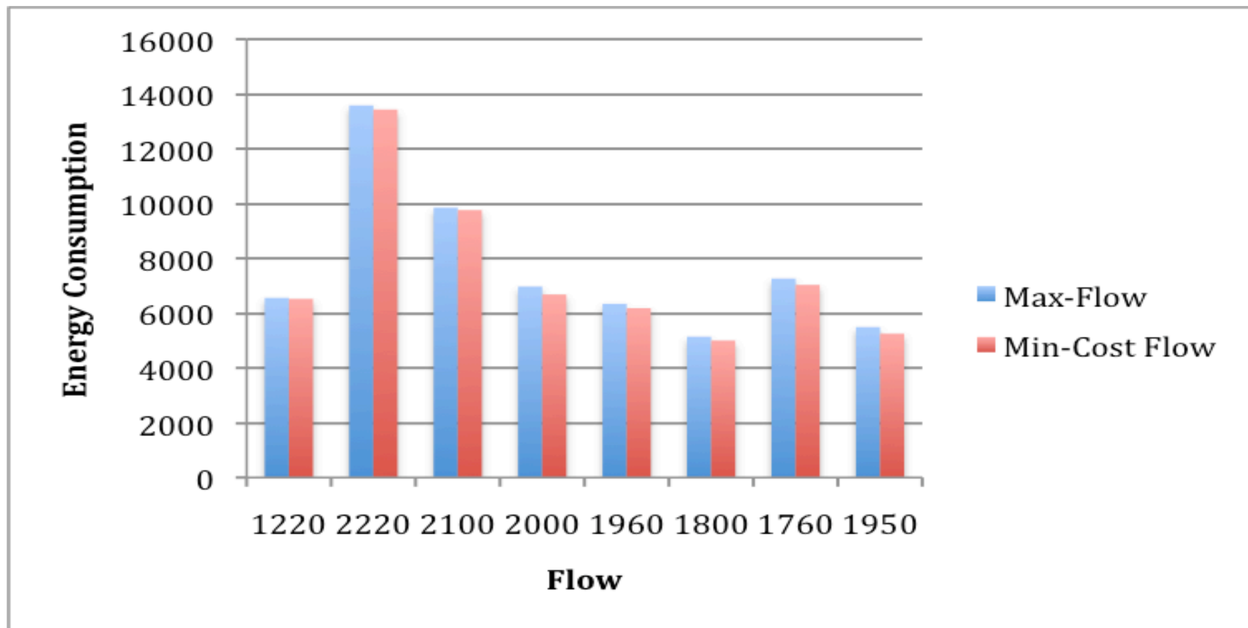

Fig. 15. Energy consumption vs Flow for a 15x15 network for varying data generators from 1 to 15.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 CONCLUSION

We study the data preservation problem in wireless sensor networks in two stages. In first stage, we showed that maximizing the data preservation time is possible while still minimizing the cost of redistribution. We prove that the load-balancing data preservation algorithm performs better than the minimum cost flow algorithm. In the second stage, we consider limited battery power in each sensor node and show that minimizing the total energy consumption for data preservation is still feasible. And then we showed that the energy consumption from minimum cost flow algorithm is always greater than or equal to the energy consumption from maximum flow algorithm.

5.2 FUTURE WORK

As future work, there is scope for minimizing the total energy consumption for data preservation under individual node energy and storage constraint, under general energy model instead of 0.5/0.5 energy model assumed in this thesis. Under the general energy model, if each sensor node has initial energy of E_i Joule (J), then according to the first order radio model, the transmission energy E_T to send k -bit of data over distance l on the transmitter side, is $E_T = E_{elec} k + E_{amp} k l^2$ and the receiving energy on the receiver side is $E_R = E_{elec} k$, where $E_{amp} = 100$ pJ/bit/m^2 is the energy consumption on the transmit amplifier for one bit and $E_{elec} = 100$nJ/bit is the energy consumption per bit on the transmitter and receiver circuit.

Under this energy model, the transmission energy, E_T is a function of distance between the sender and receiver and the number of data items to be offloaded whereas in this thesis it just depends on the number of data items i.e. 0.5 unit of energy to transmit one data item. In both the models the receiving energy, E_R is based on the number of data items a node receives but the difference is we use the same model of 0.5 units of energy to receive one data item and in the general model it is more precise and practicable. Accordingly, in the general model, intermediate nodes spend transmission energy, E_T and receiving energy, E_R to relay every data item but in this thesis it equals to number of data items it relays i.e. (0.5 unit of energy to receive + 0.5 unit of energy to send for each data item). Overall the general energy model is more realistic approach to consider the energy consumed at every node involved in data redistribution. To minimize the total energy consumption under this model we can transform this to minimum cost flow problem using these energy consumptions as edge costs accordingly and solve using minimum cost flow algorithm.

REFERENCES

REFERENCES

[1] Dargie, W. and Poellabauer, C., Fundamentals of wireless sensor networks: *Theory and practice, John Wiley and Sons, 2010.*

[2] http://wsnblog.com/2012/08/20/seeing-bad-smell-new-odor-sensor-available. Example of sensor devices capable of sensing odor. [Date accessed: 12 November 2012]

[3] Rex Min, Manish Bhardwaj, Seong-Hwan Cho, Eugene Shih, Amit Sinha, Alice Wang, Anantha Chandrakasan. Low-power wireless sensor networks. *Proceedings of the 14th International Conference on VLSI Design (VLSID '01) Page 205-210.*

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

[5] Bin Tang, Neeraj Jaggi, Haijie Wu, Rohini Kurkal. Energy efficient data redistribution in sensor networks. *In Proc. of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2010).*

[6] Bin Tang, Xiang Hou, Zane Sumpter, Lucas Burson, and Xinyu Xie. Maximizing Data Preservation in Intermittently Connected Sensor Networks. *In Proc. of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2012).*

[7] Maulin Patel, R. Chandrasekaran, and S. Venkatesan. Energy-efficient capacity-constrained routing in wireless sensor networks. *International Journal of Pervasive Computing and Communications*, 2(2):69-80, 2006.

[8] Andrew V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms,* 22(1):1-29, 1997.

REFERENCES

[9] http://www.igsystems.com/cs2/index.html. Implementation of minimum cost flow algorithm. [Date accessed: September 2011]

[10] http://algs4.cs.princeton.edu/64maxflow/. Implementation of maximum flow algorithm. [Date accessed: January 2012]

APPENDICES

APPENDIX A

LOAD-BALANCING DATA PRESERVATION PROGRAMS

Load-balancing data preservation algorithm explained in chapter 3 is implemented using two java programs. Main.java and LoadBalance.java. We use the implementation of minimum cost flow algorithm from [8][9]. Main.java generates the sensor network and converts into required input format for minimum cost flow program and also for load balance program. LoadBalance.java is implementation of load-balancing data preservation algorithm, it takes input files generated from Main.java and also the output from minimum cost flow algorithm and computes the minimum energy left among destination nodes from both minimum cost flow and load balance algorithm.

Main.java – This java program takes network size, number of data generators and number of data items each data generator has to offload, as input parameters and generates the complete network in required format for minimum cost flow and load-balancing program.

```
public class Main {
    private static int node_matrix;
    private static int num_nodes;
    private static int dg_arr[];
    public static void main(String[] args) throws IOException{
      Main m = new Main();
      System.out.println("Enter the size of the grid network
      (Enter 5 for network size 5x5): ");
      Scanner input = new Scanner(System.in);
      node_matrix = input.nextInt();
```

```
num_nodes = node_matrix * node_matrix;

System.out.println("Enter the number of the data
generators:");

Scanner input_dg = new Scanner(System.in);

dg_arr = new int[input_dg.nextInt()];

System.out.println("Enter the number of the data items for
each data generator:");

Scanner input_dataitems = new Scanner(System.in);

int data_items = input_dataitems.nextInt();

//Specify which node to be the data generator,

int dg = 1;

for(int i=0;i<dg_arr.length;i++){

      dg_arr[i] = dg;

      dg = 55;}

write_network_to_file(dg_arr,"dg_arr.txt");

int x1[] =  new int[node_matrix];

int y1[] =  new int[node_matrix];

int count = 1;

int x2[] = new int[dg_arr.length];

int y2[] = new int[dg_arr.length];

int network[][] = new int[node_matrix][node_matrix];

for(int i=0; i< node_matrix ; i++){

      x1[i] = i;

      for(int j=0; j< node_matrix ; j++){

          y1[j] = j;

          network[i][j] = count;
```

```
            for(int k=0 ; k <dg_arr.length ;k++){
                    if(count == dg_arr[k]){
                     x2[k] = i;
                     y2[k] = j;
                }
            }
            count++;
        }
    }
/* Assign node number to all the sensors */
int sensor[] = new int[num_nodes - dg_arr.length];
int sen_index =0;
for(int i=1;i<=num_nodes;i++){
     int count2=0;
     for(int k=0; k<dg_arr.length; k++){
          if(i != dg_arr[k]){
               count2++;
               if(count2 == dg_arr.length){
                    sensor[sen_index] = i;
                    sen_index++;
               }
          }
     }
 }
```

```
    /* Assigns energy to all the sensor nodes */

    int[] energyArr = assign_energy(dg_arr, x1, y1, x2, y2,
    node_matrix, sensor);

    // Find the distance between DG and every sensor node
    and print them in the output file

    int dist = 0;

    for(int k=0 ; k<dg_arr.length ;k++){

        for(int i=0; i<node_matrix ; i++){

            for(int j=0; j<node_matrix ; j++){

                if(network[i][j] != dg_arr[k]){

                    dist = Math.abs(x2[k]-x1[i])+
                          Math.abs(y2[k]-y1[j]);

                }

             }

        }

    }

    /* Prints the network details to input file for CS2.c */

    write_to_output_file(dg_arr,data_items,x1, y1, x2,
    y2,node_matrix, sensor);

    write_array_to_file(energyArr,"eninput.txt");

    write_to_output_file_single_line(dg_arr,data_items,x1,
    y1, x2, y2,node_matrix, sensor);

}
```

```
/**Function to write the network to file: output.inp,    which
will be the input to minimum cost flow program*/

public   static   void   write_to_output_file(int   dg_arr[],int
dataItems,int  x1[],  int  y1[],  int  x2[],    int  y2[],int
node_matrix,int sensor[]) throws IOException{

       int num_nodes = node_matrix * node_matrix;

       int          num_edges          =          dg_arr.length
        *sensor.length+dg_arr.length+sensor.length;

       int source_node = num_nodes +1;

       int sink_node = num_nodes +2;

        int data_items = dg_arr.length*dataItems;

        try{

            Writer output = null;

            File file = new File("output.inp");

            output = new BufferedWriter(new FileWriter(file));

            int total_nodes = num_nodes+2;

            output.write("p   min   "   +   total_nodes   +"   "+
            num_edges);

            output.write("\n");

            output.write("c   min-cost   flow   problem   with
            "+num_nodes+ " nodes and " +num_edges+" arcs");

            output.write("\n");

            output.write("n "+source_node +" "+ data_items);

            output.write("\n");

            output.write("c supply of "+ data_items + " at node
            "+ source_node);

            output.write("\n");

            output.write("n "+sink_node +" "+ -data_items);
```

```
output.write("\n");

output.write("c demand of "+ -data_items + " at node
"+ sink_node);

output.write("\n");

output.write("c edge details follows");

output.write("\n");

//Print the edge details from source to Data
Generators

output.write("c edge details from source to Data
Generators");

output.write("\n");

for(int k=0; k<dg_arr.length; k++){

  output.write("a "+source_node+ " "+ dg_arr[k] + "
  0 "+ dataItems +" 0");

  output.write("\n");

}

//Write the edge details from sensor nodes to sink

output.write("c edge details from sensor nodes to
 sink");

output.write("\n");

for(int i=0; i<sensor.length;i++){

    output.write("a "+sensor[i]+ " "+ sink_node+ " 0
    10  0");

    output.write("\n");

}

int dist = 0;

for(int k=0 ; k <dg_arr.length ;k++){
```

```
        int count1 = 1;
        for(int i=0; i<node_matrix ; i++){
          for(int j=0; j<node_matrix ; j++){
              for(int y=0; y<sensor.length; y++){
                  if(count1 == sensor[y]){
                      dist    =   Math.abs(x2[k]-x1[i])    +
                      Math.abs(y2[k]-y1[j]);
                      output.write("a  "+  dg_arr[k]+  "  "+
                      count1 + " 0 10 " + dist);
                      output.write("\n");
                  }
              }
              count1++;
          }
        }
     }
     output.close();
     }catch(IOException ioe){
            System.out.println(ioe.getMessage());
            ioe.printStackTrace();
     }
}
```

```
/**Function to write network to file: network.txt that is used
by load balancing program*/

public static void write_to_output_file_single_line(int
dg_arr[],int dataItems,int x1[], int y1[], int x2[], int
y2[],int node_matrix,int sensor[]) throws IOException{

        int num_nodes = node_matrix * node_matrix;

        int num_edges = dg_arr.length
        *sensor.length+dg_arr.length+sensor.length;

        int source_node = 0;

        int sink_node = num_nodes + 1;

        int data_items = dg_arr.length*dataItems;

        try{

            Writer output = null;

            File file = new File("network.txt");

            output = new BufferedWriter(new FileWriter(file));

            int total_nodes = num_nodes+2;

            output.write(total_nodes +" "+ num_edges);

            //Print the edge details from source to Data
            Generators

            for(int k=0; k<dg_arr.length; k++){

               output.write(" "+source_node+ " "+ dg_arr[k] + "
             "+ dataItems +" 0");

            }

            //Print the edge details from sensor nodes to sink
            node

            for(int i=0; i<sensor.length;i++){

                    output.write(" "+sensor[i]+ " "+ sink_node+
                    " 10 0");

            }
```

```
        /*Edges between neighbor nodes*/
        int dist = 0;
        for(int k=0 ; k <dg_arr.length ;k++){
            int count1 = 1;
            for(int i=0; i<node_matrix ; i++){
                for(int j=0; j<node_matrix ; j++){
                    for(int y=0; y<sensor.length; y++){
                        if(count1 == sensor[y]){
                            dist    =    Math.abs(x2[k]-x1[i])    +
                            Math.abs(y2[k]-y1[j]);
                            output.write("   "+dg_arr[k]+   "   "+
                            count1 + " 10 " + dist);
                        }
                    }
                    count1++;
                }
            }
        }
        output.close();
        }catch(IOException ioe){
               System.out.println(ioe.getMessage());
               ioe.printStackTrace();
      }
    }
```

```
/* Assigns energy to all the sensor nodes */

public static int[] assign_energy(int dg_arr[],int x1[], int
y1[], int x2[], int y2[],int node_matrix,int sensor[]){

        int energyArr[] = new int[node_matrix*node_matrix];

        int count =0;

        for(int i=0; i< node_matrix; i++){

            for(int j=0; j<node_matrix; j++){

                energyArr[count] = 1000;

                count++;

            }

        }

        return energyArr;

}

/**Function to write the initial energy levels to file:
eninput.txt */

public static void write_array_to_file(int energyArr[], String
filename) throws IOException{

      try{

        Writer output = null;

        File file = new File(filename);

        output = new BufferedWriter(new FileWriter(file));

        for(int k=0; k<energyArr.length; k++){

             output.write(""+energyArr[k]);

             output.write("\n");

       }

        output.close();
```

```
        }catch(IOException ioe){
                System.out.println(ioe.getMessage());
                ioe.printStackTrace();
        }
}
/**Function to write the data generator details to the given
file name */
public static void write_network_to_file(int dg_arr[], String
filename) throws IOException{
    try{
       Writer output = null;
       File file = new File(filename);
       output = new BufferedWriter(new FileWriter(file));
       output.write(""+node_matrix);
       output.write("\n");
       output.write(""+dg_arr.length);
       output.write("\n");
       for(int k=0; k<dg_arr.length; k++){
             output.write(""+dg_arr[k]);
             output.write("\n");}
             output.close();
       }catch(IOException ioe){
               System.out.println(ioe.getMessage());
               ioe.printStackTrace();}
       }
   }
```

LoadBalance.java: It takes network.txt, dg_arr.txt generated by Main.java and Output.txt, output from minimum cost flow program, as input files and performs load-balancing on the given network and outputs the minimum energy left among the destination nodes for both minimum cost flow and load-balancing data preservation algorithm post redistribution.

```
import java.util.*;
import java.io.*;
import java.util.ArrayList;
public class LoadBalance {
    private int node_matrix;
    private int dgNo;
    private int dg_arr[];
    private   int x1[] ;
    private   int y1[] ;
    private   int count = 1;
    private   int x2[];
    private   int y2[];
    private   int network[][];
    double[] energyArr;
    double[] energyArr2;
    ArrayList<Integer> destinationNode=new ArrayList<Integer>();
    double[] energyOfdestNodes;
    double[] energyOfdestNodes2;
    ArrayList <Path>  paths = new ArrayList<Path>();
```

```
Pathtest pathtest ;
public static void main(String[] args)  throws IOException{
    //Read the output from Minimum-cost flow program
    LoadBalance lb = new LoadBalance();
    lb.createNetwork();
    lb.read_file_from_cs2_program();
    lb.printRemainingEnergy();
}
public void printRemainingEnergy(){
    energyOfdestNodes = new double[destinationNodes.size()];
    energyOfdestNodes2= new double[destinationNodes.size()];
    //Print the least energy from Minimum-cost flow program
    int k=0;
    for(int j=0;j<destinationNodes.size();j++){
        for(int i=0;i<energyArr.length;i++){
            if(destinationNodes.get(j).intValue()-1 == i){
                energyOfdestNodes[k] = energyArr[i];
                k++;
            }
        }
    }
    System.out.println ("Least Energy from Minimum Cost Flow
     :"+get_node_with_minimum_energy(energyOfdestNodes));
    //Print the least energy from Load Balance
    int m=0;
```

```
    for(int j=0;j<destinationNodes.size();j++){
        for(int i=0;i<energyArr2.length;i++){
            if(destinationNodes.get(j).intValue()-1 == i){
                energyOfdestNodes2[m] = energyArr2[i];
                m++;
            }
        }
    }
    System.out.println ("Least Energy from Load Balance
    :"+get_node_with_minimum_energy(energyOfdestNodes2));
}
public  void read_file_from_cs2_program(){
 String minCostoutputFile = "OutputFromMinimumCostFlow.txt";
 try{
     FileInputStream fstream = new
     FileInputStream(minCostoutputFile);
     // Get the object of DataInputStream
      DataInputStream in = new DataInputStream(fstream);
      BufferedReader br = new BufferedReader(new
      InputStreamReader(in));
      String strLine;
      //Read File Line By Line
      while ((strLine = br.readLine()) != null)   {

      if(strLine.startsWith("f")&&((!strLine.endsWith("0")||(
      (strLine.endsWith("10"))))){
         for(int i=0;i<dg_arr.length;i++){
```

```
                int dataItems = 0;
                String str = "f       "+dg_arr[i];
                String str1 = "f      "+dg_arr[i];

                if((strLine.startsWith(str))||(strLine.startsWi
                th(str1))){
                  int                   nonDgNode              =
                  Integer.parseInt(strLine.substring(14,
                  17).trim());
                  dataItems                                    =
                  Integer.parseInt(strLine.substring(26,28).tri
                  m());
                  loadBalance(dg_arr[i],nonDgNode,dataItems,i);
                }
              }
            }
          }
          //Close the input stream
          in.close();
            }catch (Exception e){//Catch exception if any
              System.err.println("Error: " + e.getMessage());
          }
}
public void createNetwork(){
      String dgFile = "dg_arr.txt";
      try{
        FileInputStream          fstream          =          new
        FileInputStream(dgFile);
```

```
// Get the object of DataInputStream
DataInputStream in = new DataInputStream(fstream);
BufferedReader    br    =    new    BufferedReader(new
InputStreamReader(in));
String strLine;
String nodeMatrix = br.readLine();
node_matrix = Integer.parseInt(nodeMatrix);
energyArr = new double[node_matrix*node_matrix];
energyArr2 = new double[node_matrix*node_matrix];
String dg_size = br.readLine();
dgNo = Integer.parseInt(dg_size);
x1 =  new int[node_matrix];
y1 =  new int[node_matrix];
x2 = new int[dgNo];
y2 = new int[dgNo];
network = new int[node_matrix][node_matrix];
dg_arr  = new int[dgNo];
int i=0;
//Read File Line By Line
while ((strLine = br.readLine()) != null)   {
    int dg = Integer.parseInt(strLine);
    dg_arr[i] = dg;
    i++;
}
//Close the input stream
in.close();
```

```
  }catch (Exception e){//Catch exception if any
       System.err.println("Error: " + e.getMessage());
}
for(int i=0; i< node_matrix ; i++){
    x1[i] = i;
    for(int j=0; j< node_matrix ; j++){
        y1[j] = j;
        network[i][j] = count;
        for(int k=0 ; k <dgNo ;k++){
            if(count == dg_arr[k]){
               x2[k] = i;
               y2[k] = j;
            }
        }
        count++;
    }
}
String energyFile = "eninput.txt";
try{
    FileInputStream fstream = new
    FileInputStream(energyFile);
    // Get the object of DataInputStream
    DataInputStream in = new DataInputStream(fstream);
    BufferedReader br = new BufferedReader(new
    InputStreamReader(in));
    String strLine;
```

```
        int i=0;
        //Read File Line By Line
        while ((strLine = br.readLine()) != null)   {
                int energy = Integer.parseInt(strLine);
                energyArr[i] = energy;
                i++;
        }
        //Close the input stream
        in.close();
    }catch (Exception e){//Catch exception if any
        System.err.println("Error: " + e.getMessage());
    }
    for(int x=0;x<energyArr.length;x++)
       energyArr2[x] = energyArr[x]
}
public void loadBalance(int source, int dest, int dataItems,int
index){
      pathtest = new Pathtest(network);
      //Get the co-ordinates of non-dg node
      //Get the co-ordinates of destination node with maximum
       energy
      int dest_x2 = 0;
      int dest_y2 = 0;
          for(int i=0; i< node_matrix ; i++){
              for(int j=0; j< node_matrix ; j++){
                  if(network[i][j] == dest){
```

```
                    dest_x2 = x1[i];
                    dest_y2 = y1[j];
                }
            }
    }
    //For every data item from dg to non-dg node, find all
      the shortest path and do the node balance
    //Find shortest path with highest least energy and then
      update the energy
    paths = pathtest.findPath(x2[index], y2[index], dest_x2,
    dest_y2);
    int pathCount = paths.size();
    for(Path p: paths){
          for(int n=0; n<p.pathAsArray().size();n++){
            int intDestNode = p.pathAsArray().get(n);
            if(intDestNode != source)
          destinationNodes.add(Integer.valueOf(intDestNode));
          }
    }
    if(pathCount > 1){
    Random randomGen =  new Random();
    for(int x=1;x<=dataItems;x++){
       // Find all the shortest paths to the destination node
         double leastEnergyArr[] = new double[pathCount];
         int y=0;
         for(Path p : paths){
```

```
    double          energyAtNode[]          =          new
    double[p.pathAsArray().size()-2];

    int v=0;

    //Get the energy of all the nodes in this path in an
     array

    for(int n=0; n<p.pathAsArray().size();n++){

            int intDestNode = p.pathAsArray().get(n);

            if((intDestNode != dest ) && (intDestNode !=
            source) ){

                energyAtNode[v]  =  energyArr2[intDestNode
                - 1];

                v++;

             }

     }

     leastEnergyArr[y]=
     get_node_with_minimum_energy(energyAtNode);

     y++;

}

//Get path with highest least energy and offload the data
  item

Path optimumPath =  get_node_max_energy(leastEnergyArr);

for(int z=0; z<optimumPath.pathAsArray().size();z++){

   int intDestNode = optimumPath.pathAsArray().get(z);

   if(energyArr2[intDestNode-1] >= 1){

      if((intDestNode  ==  dest  )  ||  (intDestNode  ==
      source) ){

        energyArr2[intDestNode-1]                      =
        energyArr2[intDestNode-1] - 0.5;

        }else{
```

```
                    energyArr2[intDestNode-1] =
                    energyArr2[intDestNode-1] - 1;

                }

            }

        }

        //Get a random path to offload every data item

        Path randomPath =
        paths.get(randomGen.nextInt(paths.size()));

        for(int z=0; z<randomPath.pathAsArray().size();z++){

            int intDestNode = randomPath.pathAsArray().get(z);

            if(energyArr[intDestNode-1] >= 1){

              if((intDestNode == dest ) || (intDestNode ==
              source) ){

                 energyArr[intDestNode-1] =
                 energyArr[intDestNode-1] - 0.5;

              }else{

                 energyArr[intDestNode-1] =
                 energyArr[intDestNode-1] - 1;

              }

            }

         }

       }

    }

}

public double get_node_with_minimum_energy(double energyArr[]){

         double temp = Integer.MAX_VALUE;

         for(int i=0; i< energyArr.length;i++){
```

```
             if(energyArr[i] < temp){
                temp = energyArr[i];
            }
        }
        return temp;
 }
 /* Returns the node with maximum energy */
 public Path get_node_max_energy(double[] energyArr){
        double temp = 0;
        int index = 0;
        for(int i=0; i< energyArr.length;i++){
            if(energyArr[i] > temp){
                temp = energyArr[i];
                index = i;
            }
        }
        return paths.get(index);
  }
}
```

APPENDIX B

PROGRAM TO GENERATE NETWORK FOR MINIMUM ENERGY DATA PRESERVATION

NetworkGenerator.java – This java program takes network size, number of data generators and number of data items, each data generator has to offload, as input from user and generates complete network in required format for minimum cost flow and maximum flow program. The generated network is according to graph shown in Figure 6.2 for the user-input grid network. We use implementation of minimum cost flow algorithm from [9] and implementation of maximum flow program from [10].

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.util.*;
public class NetworkGenerator {
   public static void main(String[] args) {
        //Get the network size from the user
        System.out.println("Enter the network size of grid
        network (Enter 5 for network size 5x5) : ");
        Scanner in = new Scanner(System.in);
        int networkSize = in.nextInt();
        //Determine number of nodes based on network size
        int totalNodes = networkSize*networkSize;
        System.out.println("Enter the number of Data Generators:
        ");
        int numSourceNodes = in.nextInt() ;
        System.out.println("Enter the data items for each DG:
        ");
```

```
int dataItems = in.nextInt() ;

/* Assign energy, capacity to all nodes and split the

 * nodes, for Split node network, Every node has it
 split node, which is numbered as: (node number*100) as

 * reference and it is called as double prime node from
 */

ArrayList<Node> nodes = new ArrayList<Node>();

for(int i=1; i<=totalNodes; i++){

   Node n = new Node(i,300,10, i, i*100);

   nodes.add(n);

}

System.out.print("\n");

System.out.println("Energy and Capacity of all the nodes
:");

//Print the network generated

for (Node node : nodes) {

    System.out.println("Node: " +node.getNodeNumber()

            + " Energy: "+ node.getEnergy()

            + " Capacity: "+ node.getStorage());

}

System.out.print("\n");

//Selecting randomly the dg nodes and set high energy

ArrayList<Node> sourceNodes = new ArrayList<Node>();

for(int i=1;i<=numSourceNodes;i++){

    Node                       sn                        =
    nodes.get(generate_random_number(networkSize*network
    Size));
```

```
        sn.setEnergy(10000);

        sn.setSourceNode(true);

        sourceNodes.add(sn);

    }

    System.out.println("The randomly chosen data generators
    are:");

    for (Node node : sourceNodes) {

         System.out.println("Data        Generator:         "
        +node.getNodeNumber()+ " Energy: "+ node.getEnergy()

         + " Capacity: "+ node.getStorage() );

    }

    //Assign x and y co-ordinates for all nodes

    int n = 0;

    for(int i=0; i<networkSize;i++){

         for(int j=0; j<networkSize;j++){

            nodes.get(n).setXCor(i);

            nodes.get(n).setYCor(j);

            n++;

        }

    }

    //Splitting the nodes and creating edges between them

    ArrayList<Edge> splitNodeEdges = new ArrayList<Edge>();

    for(Node node:nodes){

       Edge                e                =                new
       Edge(node.getPrimeNode(),node.getDoublePrimeNode(),no
       de.getStorage(),0);

       splitNodeEdges.add(e);
```

```
}
//Create edges between neighbor nodes in a grid network
int ind =0;
ArrayList<Edge> neighborEdges = new ArrayList<Edge>();
for(int ne=0;ne<nodes.size();ne++){
   Node node = nodes.get(ne);
   ind = ne-networkSize;
   if(ind >=0 && ind < totalNodes)
   {
      Node firstVerticalNode = nodes.get(ind);
      int  dist  =  Math.abs(firstVerticalNode.getXCor()-
      node.getXCor())+Math.abs(firstVerticalNode.getYCor
      ()-node.getYCor());
      if(dist == 1){
      Edge         e          =          new          -
      Edge(node.getDoublePrimeNode(),firstVerticalNode.g
      etPrimeNode(),  10000, 1);
      if(!(neighborEdges.contains(e))){
         neighborEdges.add(e);
      }
      Edge            e1             =              new
      Edge(firstVerticalNode.getDoublePrimeNode(),
      node.getPrimeNode(), 10000, 1);
      if(!(neighborEdges.contains(e1))){
                neighborEdges.add(e1);
      }

    }
```

```
}

ind = ne+networkSize;

if(ind >=0 && ind < totalNodes)

{

   Node secondVerticalNode = nodes.get(ind);

   int  dist  =  Math.abs(secondVerticalNode.getXCor()-
   node.getXCor())+Math.abs(secondVerticalNode.getYCo
   r()-node.getYCor());

   if(dist == 1){

      Edge              e              =              new
      Edge(node.getDoublePrimeNode(),secondVerticalNo
      de.getPrimeNode(),  10000, 1);

      if(!(neighborEdges.contains(e))){

              neighborEdges.add(e);

      }

      Edge              e1              =              new
      Edge(secondVerticalNode.getDoublePrimeNode(),
      node.getPrimeNode(), 10000, 1);

      if(!(neighborEdges.contains(e1))){

             neighborEdges.add(e1);

       }

    }

}

ind = ne-1;

if(ind >=0 && ind < totalNodes)

{

   Node firstHorizontalNode = nodes.get(ind);
```

```
   int dist = Math.abs(firstHorizontalNode.getXCor()-
   node.getXCor())+Math.abs(firstHorizontalNode.getYC
   or()-node.getYCor());

   if(dist == 1){

     Edge              e              =              new
     Edge(node.getDoublePrimeNode(),firstHorizontalNo
     de.getPrimeNode(),  10000, 1);

     if(!(neighborEdges.contains(e))){

              neighborEdges.add(e);

     }

     Edge              e1             =              new
     Edge(firstHorizontalNode.getDoublePrimeNode(),
     node.getPrimeNode(), 10000, 1);

     if(!(neighborEdges.contains(e1))){

             neighborEdges.add(e1);

     }

   }

}

ind = ne+1;

if(ind >=0 && ind < totalNodes)

{

   Node secondHorizontalNode = nodes.get(ind);

   int                    dist                    =
   Math.abs(secondHorizontalNode.getXCor()-
   node.getXCor())+Math.abs(secondHorizontalNode.getY
   Cor()-node.getYCor());

   if(dist == 1){

      Edge             e              =              new
      Edge(node.getDoublePrimeNode(),secondHorizontal
      Node.getPrimeNode(),  10000, 1);
```

```
                    if(!(neighborEdges.contains(e))){

                       neighborEdges.add(e);

                    }

                    Edge             e1             =            new
                    Edge(secondHorizontalNode.getDoublePrimeNode(),
                    node.getPrimeNode(), 10000, 1);

                    if(!(neighborEdges.contains(e1))){

                           neighborEdges.add(e1);

                    }

                  }

      }}

      //Transform network into desired graph and write it to a file

      //Create input file for cs2 program

      write_network_to_file(dataItems,           nodes,sourceNodes,
      splitNodeEdges, neighborEdges);

      //Create input file for max flow program

      write_to_output_file_single_line(dataItems,
      nodes,sourceNodes, splitNodeEdges, neighborEdges);

   }

   public static int generate_random_number(int max){

           int energy = 0;

           int min = 1;

           energy = min + (int)(Math.random() * ((max - min)+1));

           return energy;

    }
```

```
public static Node get_node_with_max_energy(ArrayList<Node> nodes){
        int max_energy = 0;
        Node nodeWithMaxEnergy = null;
        for(Node tempNode: nodes){
            if(tempNode.getEnergy() > max_energy){
                max_energy = tempNode.getEnergy();
                nodeWithMaxEnergy = tempNode;
            }
        }
        return nodeWithMaxEnergy;
}
/*
 * This function will transform a network into a graph, adding super source and super nodes
 * Adding capacities, creating edges and writes the created network to a file
 */
public static void write_network_to_file(int dataItems, ArrayList<Node> nodes, ArrayList<Node> sourceNodes, ArrayList<Edge> splitEdges,ArrayList<Edge> neighborEdges){
    try{
            //Create file
            FileWriter OutStream = new FileWriter("Min-Cost-Input.inp");
            BufferedWriter out = new BufferedWriter(OutStream);
            out.write("c Input details start below ");
            out.newLine();
```

```
/*Get total no of nodes and edges, the total number of nodes
 *has to be equal or greater than the highest split node's
 *number.For ex in a 10*10 network, for node 100 its split node
 *is 10000, so taking 10000 for total number of nodes.Just for
 *reference, super source is 555 and super node is 999 */

int edges =
sourceNodes.size()+splitEdges.size()+neighborEdges.size()+(node
s.size()-sourceNodes.size());

out.write("p min "+ 10000 + " "+edges);

out.newLine();

out.write("n 555 "+ dataItems*sourceNodes.size());

out.newLine();

out.write("n 999 -"+ dataItems*sourceNodes.size());

out.newLine();

//Edges between super source and source nodes

for(int i=0;i<sourceNodes.size();i++){

    out.write("a 555 "+sourceNodes.get(i).getPrimeNode()+" 0 "+
    dataItems + " 0 ");

    out.newLine();

}

//Edges between split nodes

for(int i=0;i<nodes.size();i++){

   if(nodes.get(i).isSourceNode()){

     int modifiedEnergy =
     nodes.get(i).getEnergy()+(dataItems/2);

     out.write("a "+nodes.get(i).getPrimeNode()+"
     "+nodes.get(i).getDoublePrimeNode()+" 0 "+
     modifiedEnergy+" 0 ");
```

```
      out.newLine();
    }else{
       int modifiedEnergy =
       nodes.get(i).getEnergy()+(nodes.get(i).getStorage()/2);
       out.write("a "+nodes.get(i).getPrimeNode()+"
       "+nodes.get(i).getDoublePrimeNode()
       +" 0 "+ modifiedEnergy +" 0 ");
       out.newLine();
    }
}
//Edges between neighbors
for(Edge e:neighborEdges){
    out.write("a "+ e.getHead()+" "+e.getTail()+" 0
    "+e.getCapacity()+ " "+e.getCost());
    out.newLine();
}
//Edges between super sink and all non-dg nodes
for(Node node:nodes){
   if(!node.isSourceNode()){
       out.write("a "+node.getDoublePrimeNode()+" 999 0 "+
       node.getStorage() + " 0 ");
       out.newLine();
    }
 }
 out.close();
```

```
 }catch(Exception e){
       System.err.println("Error: " + e.getMessage());
 }
}
/*
 * This function will transform given network into a graph, adding super source and super nodes
 * Adding capacities, creating edges and finally writing it to a file
 */
public static void write_to_output_file_single_line(int dataItems, ArrayList<Node> nodes, ArrayList<Node> sourceNodes, ArrayList<Edge> splitEdges,ArrayList<Edge> neighborEdges){
    try{
       //Create file
       FileWriter OutStream = new FileWriter("Max-Flow-Input.txt");
       BufferedWriter out = new BufferedWriter(OutStream);
       //Get total no of nodes and edges
       int edges = sourceNodes.size()+splitEdges.size()+neighborEdges.size()+(nodes.size()-sourceNodes.size());
       out.write( (nodes.size()*100+1) + " "+edges);
       //Edges between super source and data generators
       for(int i=0;i<sourceNodes.size();i++){
           out.write("   0   "+sourceNodes.get(i).getPrimeNode()+"   "+ dataItems + " 0");
       }
```

```
//Edges between split nodes
for(int i=0;i<nodes.size();i++){
   if(nodes.get(i).isSourceNode()){
     int                 modifiedEnergy                    =
     nodes.get(i).getEnergy()+(dataItems/2);
     out.write("          "+nodes.get(i).getPrimeNode()+"
    "+nodes.get(i).getDoublePrimeNode()
     +" "+ modifiedEnergy+" 0 ");
   }else{
    int                 modifiedEnergy                     =
    nodes.get(i).getEnergy()+(nodes.get(i).getStorage()/
    2);
    out.write("          "+nodes.get(i).getPrimeNode()+"
   "+nodes.get(i).getDoublePrimeNode()
    +" "+ modifiedEnergy +" 0 ");
  }
}
//Edges between neighbors
for(Edge e:neighborEdges){
    out.write(" "+e.getHead() +" "+e.getTail()
    +" "+e.getCapacity()+" "+ e.getCost());
}
//Edges between super sink and all non-dg nodes
for(Node node:nodes){
   if(!node.isSourceNode()){
```

```
                out.write("    "+node.getDoublePrimeNode()+"    999
               "+ node.getStorage() + " 0 ");
            }
          }
          out.close();
       }catch(Exception e){
            System.err.println("Error: " + e.getMessage());
      }
  }
}
```