5-3-2007

Distributed Algorithms for Improving Wireless Sensor Network Lifetime with Adjustable Sensing Range

Aung Aung

DISTRIBUTED ALGORITHMS FOR IMPROVING WIRELESS SENSOR NETWORK LIFETIME WITH ADJUSTABLE SENSING RANGE

by

AUNG AUNG

Under the Direction of Sushil K. Prasad

ABSTRACT

Wireless sensor networks are made up of a large number of sensors deployed randomly in an ad-hoc manner in the area/target to be monitored. Due to their weight and size limitations, the energy conservation is the most critical issue. Energy saving in a wireless sensor network can be achieved by scheduling a subset of sensor nodes to activate and allowing others to go into low power sleep mode, or adjusting the transmission or sensing range of wireless sensor nodes.

In this thesis, we focus on improving the lifetime of wireless sensor networks using both smart scheduling and adjusting sensing ranges. Firstly, we conduct a survey on existing works in literature and then we define the sensor network lifetime problem with range assignment. We then propose two completely localized and distributed scheduling algorithms with adjustable sensing range. These algorithms are the enhancement of distributed algorithms for fixed sensing range proposed in the literature. The simulation results show that there is almost 20 percent improvement of network lifetime when compare with the previous approaches.

INDEX WORDS: Wireless sensor networks, maximize lifetime, target coverage, adjustable sensing range.

DISTRIBUTED ALGORITHMS FOR IMPROVING WIRELESS SENSOR NETWORK LIFETIME WITH ADJUSTABLE SENSING RANGE

by

AUNG AUNG

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2007

DISTRIBUTED ALGORITHMS FOR IMPROVING WIRELESS SENSOR NETWORK LIFETIME WITH ADJUSTABLE SENSING RANGE

by

Aung Aung

Major Professor: Sushil K. Prasad
Committee: Yingshu Li
Raheem A. Beyah

Electronic Version Approved:

Office of Graduate Studies
College of Arts and Sciences
Georgia State University
May 2007

DEDICATION

To my beloved MOTHER**.**

ACKNOWLEDGEMENTS

First, I would like to express my gratitude to my advisor, Dr. Sushil K. Prasad, for his support and invaluable guidance throughout my study. I am also very grateful to the other members of my thesis committee, Dr. Yingshu Li and Dr. Raheem A. Beyah, for their advice and their valuable time spent in reviewing the material.

I would also like to extend my thanks to various faculty members of the Department of Computer Science, Georgia State University, for their support and assistance during my graduate studies.

I also would like to extend my appreciation to my friend, Akshaye Dhawan for having helped me in writing this thesis report and to everyone who offered me academic advice and moral throughout my graduate studies.

Finally, I would like to express my gratitude to my family, especially my mother and my wife, for their continuous encouragement and confidence in me in conducting this research project. Without their support, this research project would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

WSNs	Wireless Sensor Networks
LBP	Load Balancing Protocol for Sensing
DEEPS	Deterministic Energy-Efficient Protocol for Sensing
SNLP	Sensor Network Lifetime Problem
ALBP	Load Balancing Protocol for Adjustable range Sensing
ADEEPS	Deterministic Energy-Efficient Protocol for Adjustable range Sensing

1. INTRODUCTION

Wireless sensor networks (WSNs) are a rapidly emerging technology which will have a strong impact on research and will become an integral part of our lives in the near future. The huge application space of WSNs covers national security, surveillance, military, health care, environment monitoring and many more [1]. Due to their wide-range of potential applications, WSNs have attracted considerable research interest in recent years.

A wireless sensor network is composed of a large number of low-power, low-cost sensor nodes which are deployed close to an area of interest and are connected by a wireless interface. Sensor nodes are tiny devices equipped with sensing hardware, transceivers, processing and storage resources and batteries. In general, the sensors nodes are deployed randomly and not required to be engineered or predetermined. This allows fast random deployment in inaccessible terrains or disaster relief operations. However, this random deployment requires that sensor network protocols and algorithms must possess self organizing capabilities.

After the deployment, these nodes are generally stationary and self-organized into networks. They gather information about the monitoring region and send this information to gateway node(s) where end users can retrieve the data [1]. In this way, the sensor network provides the information and a better understanding of the monitored region. The unique power of WSNs lies in the ability to deploy a large number of tiny sensor nodes which can assemble and configure themselves as a network. The network could be easily extended by simply adding more sensor nodes with no rework or complex reconfiguration.

In contrast to traditional wireless networks, the sensor nodes in WSNs do not necessarily need to communicate directly with the nearest high power control center, but mostly with their neighboring sensor nodes and each individual sensor node becomes part of an overall infrastructure. In addition, the network can automatically adapt to compensate for node failures.

When compared with traditional ad-hoc networks, WSNs have some limitations such as limitation in power, computational capacities and memory. Sensor nodes carry limited power supply which are generally irreplaceable and may be deployed with non-rechargeable batteries. Since the sensor nodes will die one after another during the operation of the network, all the network requirements must be met with minimum power consumption due to battery limitations, and in most applications, it is impossible to replenish power resources.

In WSNs, a decrease in the number of available sensor nodes can deeply degrade the network performance or may even kill the network, as either some area is not covered or some data is not transferred through the network. Moreover, it is impossible to replace thousands of nodes in hostile or remote regions, and thus the sensor nodes needs to be utilized in an efficient manner. Another factor to be considered here is the slow improvement in battery capacities over the years [2]. Thus energy saving has become a critical issue in WSNs, and the most energy saving must to come from energy aware protocols.

The main tasks of a sensor node in a sensor network are to collect data (monitoring), perform data aggregation, and then transmit data. Among these tasks transmitting data requires much more energy than processing data [4] and the most recent efforts on optimizing the wireless sensor network lifetime have been focused on routing protocol (i.e., transmitting data to the base and data request from the base to the sensor node).

Generally, there are two approaches to the problem of saving energy in wireless sensor networks. The first one is scheduling the sensor nodes to active mode and that enables the other sensor nodes to go into low power sleep mode. The second approach is to adjust the transmission or the sensing range of the wireless sensor nodes [3]. The same example in [3] is also used here for justification. There are four sensor s_1, s_2, s_3, s_4 and three targets t_1, t_2, t_3 as shown in Fig 1.1. Each sensor has two sensing range r_1 and r_2 where $r_1 < r_2$. Fig 1.1 also provides coverage relations between sensors and targets which are: $(s_1,r_1) = \{t_3\}$, $(s_1,r_2) = \{t_1,t_3\}$, $(s_2,r_1) = \{t_2\}$, $(s_2,r_2) = \{t_1,t_2\}$, $(s_3,r_1) = \{t_2\}$, $(s_3,r_2) = \{t_2,t_3\}$, $(s_4,r_1) = \{t_1,t_3\}$, $(s_4,r_2) = \{t_1,t_2,t_3\}$, respectively. And the initial energy at each sensor E = 2, and $e_1 = 0.5$ (energy requires for one unit time with sensing range r_1) and $e_2 = 1$ (energy requires for one unit time with sensing range r_2).

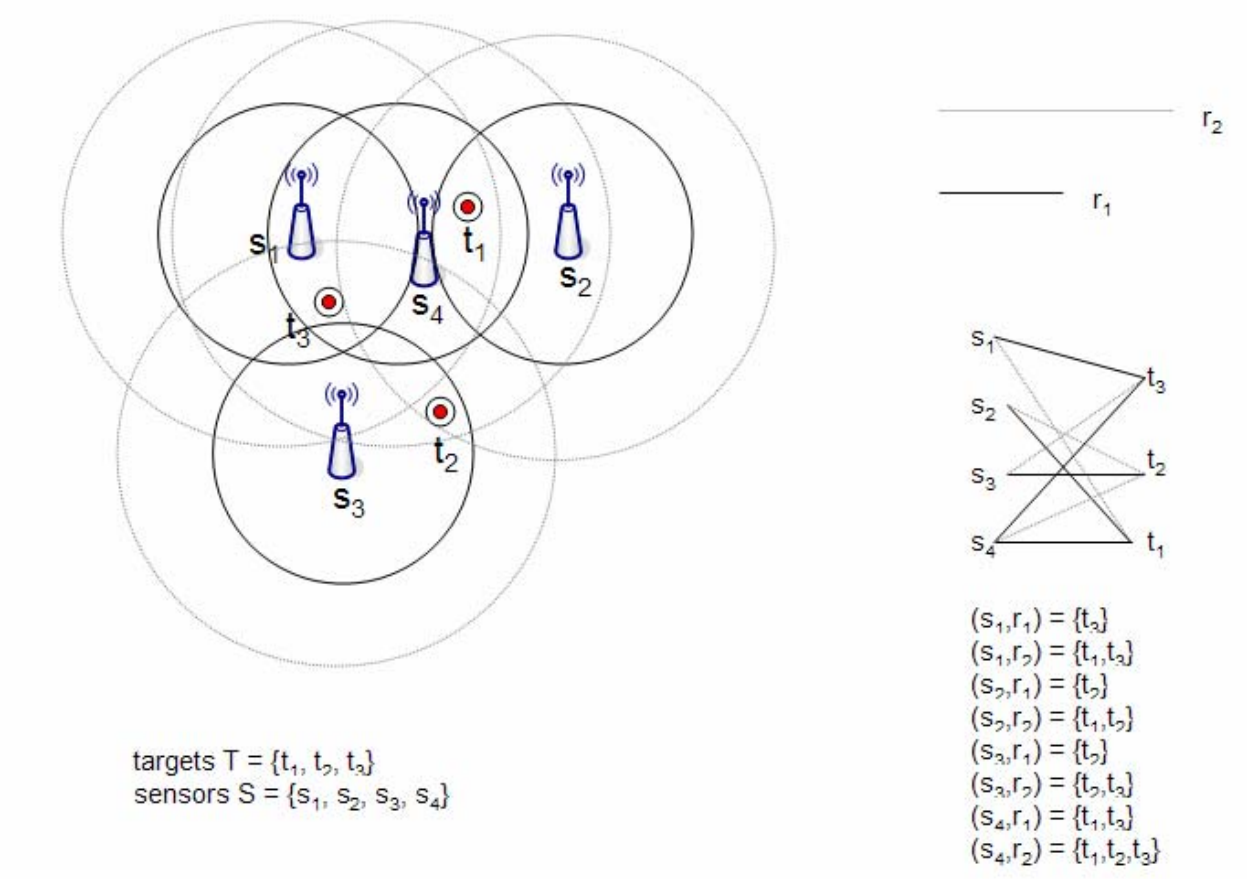


Figure 1.1 Sensor Network with four sensors and three targets, and targets covered [3]

In this sensor network, a sensor can be part of more than one cover set and five different cover sets can be obtained using the combinations of r_1 and r_2: $C_1 = \{(s_1, r_1), (s_2, r_2)\}$, $C_2 = \{(s_1, r_2), (s_3, r_1)\}$, $C_3 = \{(s_2, r_1), (s_3, r_2)\}$, $C_4 = \{(s_4, r_2)\}$, $C_5 = \{(s_1, r_1), (s_2, r_1), (s_3, r_1)\}$. These cover sets are also illustrated in Fig 1.2.

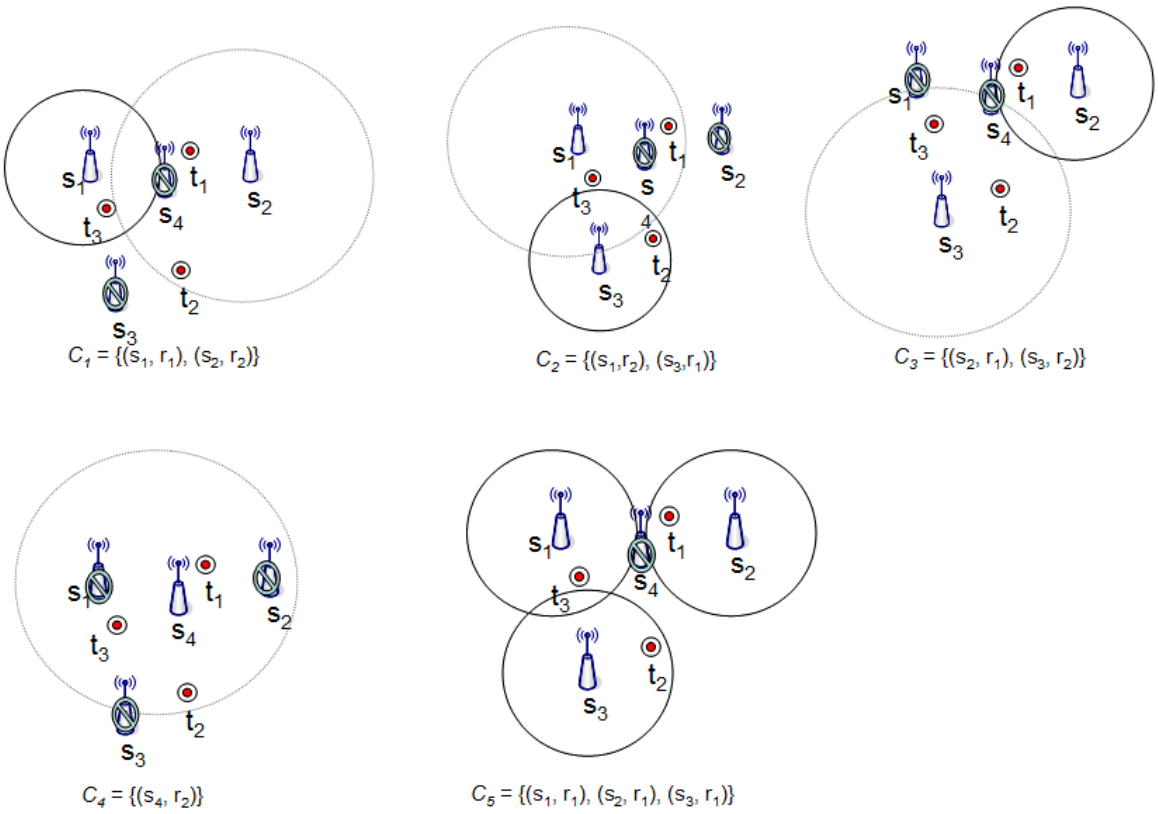


Figure 1.2 Five set covers with different sensing ranges [3].

With these five different set covers, the maximum lifetime of 6 can be obtained using the sequence as shown in the following table.

C1	**S1**=2-0.5=**1.5**	**S2** =2-1=1	
C2	**S1** =1.5 -1=**0.5**	**S3**=2-.5=**1.5**	
C3	**S2**=1-0.5=**0.5**	**S3**=1.5-1=**0.5**	
C4	**S4**=2-1=**1**		
C5	**S1**=0.5-.5=0	**S2**=0.5-0.5=0	**S3**=0.5.-0.5=0
C4	**S4**=1-1=0		

Table 1.1 Sequence of different set covers.

If the sensor nodes do not have the adjustable sensing range, sensors can be organized into two distinct set covers, such as $C_1 = \{(s_1, r_2), (s_2, r_2)\}$ and $C_4 = \{(s_4, r_2)\}$. Then the maximum life without the adjustable sensing range is 4 by using each set cover twice. Thus this example shows a 50% lifetime increase when using adjustable sensing range.

In this study, we will mostly concentrate on minimizing energy for sensing by using both smart scheduling and adjusting sensing ranges. The goal of this thesis is to maximize the lifetime of power constrained wireless sensor networks which are deployed to monitor a set of targets with known location. To maximize the lifetime, we use the property that sensors have adjustable sensing ranges, and set up the active sensors in WSNs with minimum sensing range while covering all targets.

The contributions of this thesis are two new distributed algorithms, Load Balancing Protocol for Adjustable Range Sensing (ALBP) and Deterministic Energy-Efficient Protocol for Adjustable Range Sensing (ADEEPS) are extended and implemented, and the performance of two algorithms are analyzed through simulations. The simulation results show that there is almost 20 percent improvement of network lifetime when compare with the previous approaches in [5], [6].

The organization of this thesis is as follow:

Chapter 1, “Introduction”, is an overview of what wireless sensor network is and its characteristics. It also describes the constraints in WSNs and the lifetime problems of WSNs. In addition, it provides the main goal and outline of this thesis.

Chapter 2, “Wireless Sensor Networks: Background & Related Work”, provides the literature survey of background and related works in the WSNs lifetime and coverage problems.

Chapter 3, “Network Model and Problem Definition”, describes the problem that will be solved in this thesis, and its constraints, limitations and assumptions are all addressed in this chapter.

Chapter 4, “Enhanced Distributed Algorithms”, introduces distributed LBP & DEEPS algorithms and also briefly explains ALBP and ADEEPS algorithms.

Chapter 5, “Implementation of the Algorithms”, briefly explains how the algorithms are implemented.

Chapter 6, “Simulation Results”, presents experimental setups, results and explanation. Tables and figures are provided based on the parameters used in the experiment.

Chapter 7, “Conclusions & Future Work”, summarizes the findings and contributions of this study and outlines possible directions for future research.

In addition, the algorithms designed are presented as C++ code and the sample targets and sensors files are all included in the appendices.

2. WIRELESS SENSOR NETWORKS: BACKGROUND & RELATED WORK

2.1 Background

Since the first wireless digital network, ALOHANET which was introduced in 1970, wireless networks have gained immense popularity and been growing rapidly [8]. The existing wireless network can be classified into two categories: infrastructure and ad-hoc network. The infrastructure network architecture contains a wired backbone which is connected to a special switching node or a base station. Communication between wireless hosts must go through the base station. On the other hand an ad-hoc wirelesses network is a self-organizing system with wireless hosts that do not depend on any fixed network infrastructure.

The wireless sensor networks are new families of wireless networks which comprise of many sensor nodes. After deployment, these nodes are generally stationary and self-organized into a network. They perform sensing tasks and send the information to a control center or base station, where the end-user can retrieve the data.

Although wireless sensor networks are similar to wireless data network in some aspect, their functions also need to perform in a different manner as they are performing different tasks. The dynamic and self organizing nature of wireless sensor networks require new generation of protocols which are designed to handle the immense amount of data generated by sensors and to do so with very little power.

2.2 Issues and Challenges

Wireless sensor network design is influenced by many factors, which include fault tolerance, scalability, production costs, operating environment, network topology, hardware constraints, transmission media, and power consumption [1]. These factors are important because they serve as a guideline to design a protocol or an algorithm for wireless sensor networks.

The architecture of sensor hardware consists of the usual components like processor, memory, wireless interface, power supply, as well as the sensing devices [9]. However the computational and energy resources are limited due to size and weight restriction. Wireless sensor networks are formed dynamically and contain very large number of sensors nodes. They will be deployed spontaneously to form efficient ad-hoc networks using sensors with limited computational, storage and short-range wireless communication capabilities.

Sensor devices may be deployed in very harsh environments, and subject to destruction and dynamically changing conditions. The configuration of the network will frequently change due to constant changes in accessibility of sensors, power availability and task requirements. The network protocols must be survivable in spite of device failures and frequent real-time changes.

Among these factors, limited power supply is the most critical issue due to the battery size and weight limitation. Thus the efficient use of available energy resources directly impacts the lifetime and performance of wireless sensor networks and it is very important that the algorithms/protocols used in wireless sensor networks must optimize the sensor energy utilization [10].

A sensor node can be in four states of communication: transmit, receive, idle, or sleep and two states in monitoring: idle and active [6]. The sensor network lifetime can be improved by allowing some sensors to sleep while other sensors are covering the area/target of interest. Other power saving techniques include power controlling by adjusting the sensing/transmitting range of sensor, using the energy efficient routing and data gathering techniques, reducing the amount of data transmitted and avoiding useless activity [10].

2.3 Related Work

In this section, we will highlight the previous works on improving the lifetime of wireless sensor networks by using various scheduling algorithms and adjusting sensing range of sensors.

Various centralized and distributed scheduling algorithms have been proposed in literature [16], [17], [18], [19], [20]. The coverage problem is introduced in [7] and it can be classified into three groups: area coverage (where the objective is to cover an area), target coverage (where the objective is to cover a set of targets) and breach coverage (where the objective is to find out the maximal support/breach path that traverses a sensors field) [3]. The goal of the coverage problem is to maximize the network life time while covering the sets of targets/area.

In [21], the authors introduce the area coverage with adjustable sensing range sensors. M. Cardei et al. introduce target coverage problem in [3] where disjoint sensor sets are modeled as disjoint set covers so that every cover set completely monitors all the target points. These sensor sets can be scheduled to activate successively so that at any time, one sensor set is in active state and other sensors are in sleep state. These alternations increase the lifetime of the network and

consequently, decrease the density of active nodes thus reducing the contention at the MAC layer [10]. It has also been proven in [13] that the disjoint set coverage problem is NP-complete and has a lower approximation bound of 2 for any polynomial time approximation.

In [5], [14], [10], the disjoint set cover problem is further extended by not requiring the sensor sets to be disjoint (i.e., a sensor can be active in more than one sensor set) thereby, allowing the sets to operate for different time intervals. It is also shown in [10] that non-disjoint sensor covers can provide better lifetime when compared to disjoint set covers.

In [12], the authors introduce algorithms where each sensor can produce a number of schedules which are exchanged with the neighboring sensors and the most suitable scheduled is then selected. These algorithms are analyzed through simulations. In [5], good centralized approximation algorithms as well as distributed algorithms are given, and a similar centralized approximation algorithm is also proposed in [10].

The algorithm proposed in [6] is the extension and refinement of distributed algorithm in [5]. All the algorithms in [5], [6], [10], [12], and [22] are studied for fixed sensing range sensor networks. With reference to [21], [26], sensors with adjustable sensing rages are commercially available. Cardei et al. introduce the problem of maximizing the network lifetime for adjustable sensing range sensor networks [3]. They also propose efficient heuristics (both centralized and distributed) using integer programming formulation and greedy approaches.

A. Dhawan et al. propose maximization of sensor network lifetime with adjustable sensing range algorithm in [15]. It is an extension of the centralized algorithm in [5] with adjustable sensing range sensor networks. Their approach differs significantly from [3] in that they focus on maximizing the lifetime whereas [3] focuses on maximizing the number of cover sets. They also formulate their sensor networks with non-uniform batteries at the sensors and allow the sensors to vary sensing range smoothly.

They model the sensor network lifetime problem with range assignment as a linear program problem and solve the problem using Garg-Könemann algorithm [24] with approximation ratio (1+ε). This is similar to the method used in [5], [14] for solving the fixed range maximum lifetime problem. The changes made in this algorithm are that it assigns ranges to every sensor and factors in the increase in energy consumption due to the increase in distance while choosing the sensor covers.

With an adjustable sensing range sensor network, this algorithm can generate more set covers by varying the range of the sensor nodes. The sensing range can be increased to cover more targets at the cost of increasing energy consumption. This algorithm provides the best sensing range a sensor can pick to cover open targets with the consideration of increased energy consumption with distance. They also show that the lifetime problem with adjustable range assignment can be approximated within a factor of $(1+\varepsilon)(1+\ln m)$ for any $\varepsilon > 0$ with $k = O(m)$ elements to cover, where m being the number of targets.

In this thesis, we extend the distributed algorithms in [5], [6] with adjustable sensing range sensor networks.

3. NETWORK MODEL AND PROBLEM DEFINITION

In this chapter, we will explain about the network model, problem definition and assumptions. Firstly, the network model and assumptions are discussed and followed subsequently by the problem definition.

3.1 Sensor Network Model

Our network model is similar to the models described in [3], [5], [6], and [15]. We assume that sensors are deployed over the monitored region R, and each sensor s has its own monitor target i_1 where s_1 can collect the trustful data from target i_1 without the help of any other sensor. We also assume that each sensor knows its own coordinates as well as the IDs and coordinates of all the covered targets. We further extend the assumption that each sensor can also vary the sensing range smoothly.

In our network model, a sensor is either in the communication mode or monitoring mode. During communication a sensor can either be in the sleeping, listening, receiving, or sending state and during monitoring, it can either be in the idle or active state as in [6]. If the sensor is in sleeping mode, it cannot hear any packets but it can be woke up by using wakeup mechanism as in [23]. We also assume that the number of deployed sensors largely exceed the number of targets required to monitor so that some sensors can turn themselves into sleep mode and save energy.

For the correctness of monitoring protocols, we use the following requirement in the network model: for any target $t \in T$ there is at least one sensor $s \in S$ which cannot become idle unless there is an active sensor covering *t*. We also assume that each sensor can broadcast just before the battery exhaustion so that neighboring sleep node can wakeup to replace the exhausted sensor.

3.2 Problem Statement

Sensor Network Lifetime Problem with range assignment: Given a monitored region R, a set of sensors $s_1, s_2, s_3, \ldots, s_m$ and a set of targets $i_1, i_2, i_3, \ldots, i_n$ and energy supply b_i for each sensor, find a monitoring schedule (C_1, t_1), (C_2, t_2), ……, (C_k, t_k) and a range assignment for each sensor in a set C_i such that

a. $t_1 + t_2 + \ldots + t_k$ is maximized,

b. each set cover monitors all target $i_1, i_2, i_3, \ldots, i_n$, and

c. each sensor s_i does not appear in the set $C_1 \ldots C_k$ for a time more than b_i where b_i is the initial energy of sensor of s_i.

In this definition, the requirement of "$t_1 + t_2 + \ldots + t_k$ is maximized" is equivalent to maximizing network lifetime. The energy consumed by an active sensor depends on the sensing range of that sensor and if a sensor participates in more than one set, then the total energy spent is not more than *b*.

4. ENHANCED DISTRIBUTED ALGORITHMS

In this chapter, first, the two distributed algorithms LBP and DEEPS for SNLP are discussed. After that the two enhanced distributed algorithms namely, ALBP & ADEEPS, are proposed.

4.1 Load Balancing Protocol for Sensing (LBP)

This section briefly discusses the distributed load balancing protocol (LBP) with fixed sensing range. The main idea of LBP is that the maximum number of sensors are kept alive as long as possible by means of load balancing (i.e., if a certain sensor is overused compared to its neighbors, then it is allowed to sleep) [5]. In this algorithm, sensors can freely exchange active or idle states. And it is also assumed that there is no equal battery level at each sensor (if there is an equal battery level, either sensor's or target's id will be used for making a decision.). Each sensor node can be in three states:

- ***Active:*** the sensor is active and monitors the targets
- ***Idle***: idle and sleep modes, the sensor stops wasting any energy
- ***Alert***: the sensor monitors targets but will change its state to either active or idle state soon.

Each alert sensor knows all its neighbor sensors' state, i.e., any state transition is immediately broadcast with current energy supply. When a sensor is in vulnerable state, it should change its state into:

- **Active state**: if a target is solely covered by itself,
- **Idle state**: if each target covered by a sensor s is also covered either by an active sensor or a vulnerable sensor with a larger battery supply.

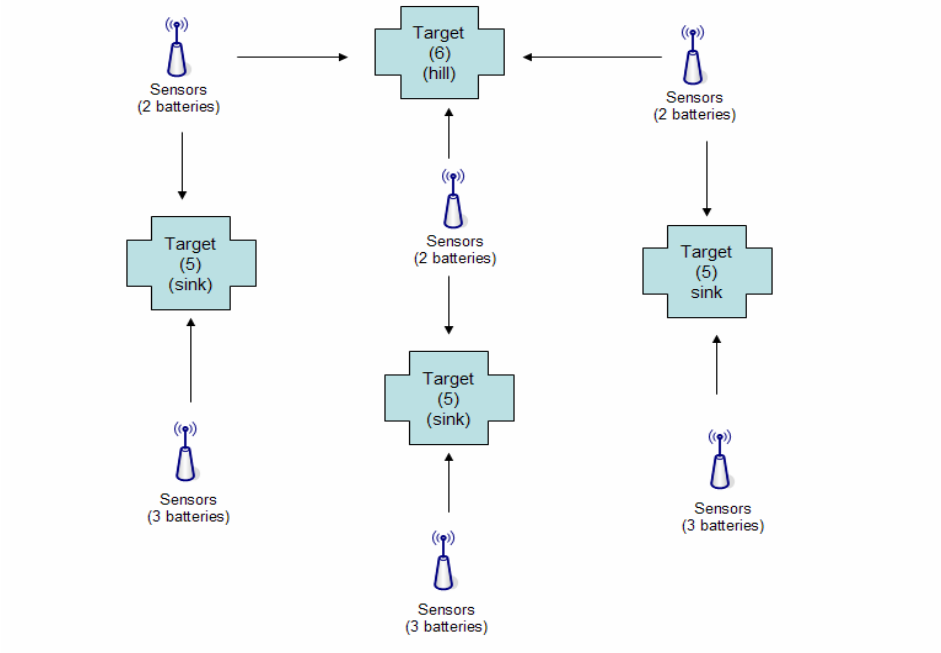
For a certain period, all nodes are altered (i.e., turn into vulnerable state) using wake-up calls, and each sensor has to decide whether to change their state to either active or idle. This process is called global reshuffle. During global reshuffle, each sensor sends two broadcasts: the first broadcast includes its cover targets and energy level and the second broadcast informs whether it will be active or idle.

If an active sensor nearly exhausts its energy, then it broadcasts about that to its neighbors. A minimal subset of neighbors in idle state will change their states into active and effectively replace the exhausted sensor. The correctness of LBP is also proved and each global reshuffle needs two broadcasts to the neighbors and the resultant set of all active sensors form a minimal sensor cover. "The main drawback of LBP is that it balances the load of sensors instead of balancing the energy of sensors covering the same target. [6]"

4.2 Deterministic Energy Efficient Protocol for Sensing (DEEPS)

This section describes the Deterministic Energy-Efficient Protocol for Sensing (DEEPS) by Dumitru et al. The main intuition behind DEEPS is that they try to minimize the energy consumption rate for low energy targets while allowing higher energy consumption for sensors with higher total supply [6].

They define each target as either "*sink*" or "*hill*". A *sink* is a target *t* which is the poorest for at least one sensor covering *t*, and the abandoned target is a *hill* which is not the poorest for any covering sensors. Each sink should be covered by a sensor with the richest batteries to keep the intuition behind LBP (keep more sensors alive). Thus DEEPS' off-rule switches off the poorer sensors covering the sinks until a single sensor switches on. This may lose the correctness of monitoring protocol requirement in which some targets may not be covered. It is shown in Fig (4.1).



Notes: The upper hill target will be uncovered if all three upper sensors with 2 batteries switch off simultaneously.

Figure 4.1 WSN with sink and hill targets [6].

For the correctness of monitoring protocol, at least one sensor is placed in-charge of each target and the sensor in-charge of *t* should not switch off until it discovers that *t* is covered by another switched-on sensor. Selection of the sensor which should be in-charge is determined by using the following two rules [6]: "(i) if the target *t* is a sink, among the sensors covering *t*, sensor *s* with the richest batteries is placed in-charge of *t*. and (ii) If target *t* is a hill, then the sensor *s* covering *t* whose poorest target is the richest over all sensors covering *t,* is placed in-charge of *t*. If there are several such sensors, then the richest among them is placed in-charge of *t*."

Consider the example in Fig 4.1, the sensors with 3 batteries are placed in-charge of sinks (i.e., three lower targets). Using the tie-breaking rule, the leftmost target is the richest among 3 lower targets. According to rule (ii), the leftmost sensor with 2 batteries will become in-charge of the topmost hill and turn on while the other 2 battery sensors will turn off.

Like LBP, sensors in DEEPS have the same states (active, idle, vulnerable) and it is necessary that all the sensors know the battery levels of all the targets of their respective neighbors, in order to calculate who is in-charge. This can be done by either sending two broadcasts or increasing the communication range. When a sensor is in vulnerable/alert state, it should change its state into:

- ***Active state***: if a target is solely covered by itself (same as LBP),
- ***Idle state***: whenever a sensor *s* is not in-charge of any target except those already covered by active sensors, *s* switches itself to idle.

The correctness of DEEPS is also proved in [6].

4.3 Load Balancing Protocol for Adjustable Range Sensing (ALBP)

Distributed Load Balancing Protocol for SNLP has been previously considered in [5]. This section describes the Load Balancing Protocol for Adjustable Range Sensing (ALBP). The objective of the monitoring protocol is to maximize the time that sensors can monitor all targets. When there is a target which cannot be covered by any sensors, the network fails. Because it is useless to have any sensors alive after the network fails, an intuition behind ALBP is to keep as many sensors alive as possible by means of load balancing and try to let them die simultaneously.

There are three main questions which should be answered by the distributed monitoring protocol for adjustable range sensors:

(1) What rules should be used to decide for each sensor node to become either idle or active?

(2) If a sensor decides to become active, what should be its sensing range?

(3) When should nodes make such decisions?

To answer these questions, we first describe the state of sensors and transition rules. At any moment, each sensor is in one of three states

- ***active***: the sensor monitors targets
- ***idle***: the sensor listens to other sensors, but does not monitor targets
- ***deciding***: the sensor monitors targets, but will change its state to either active or idle state soon

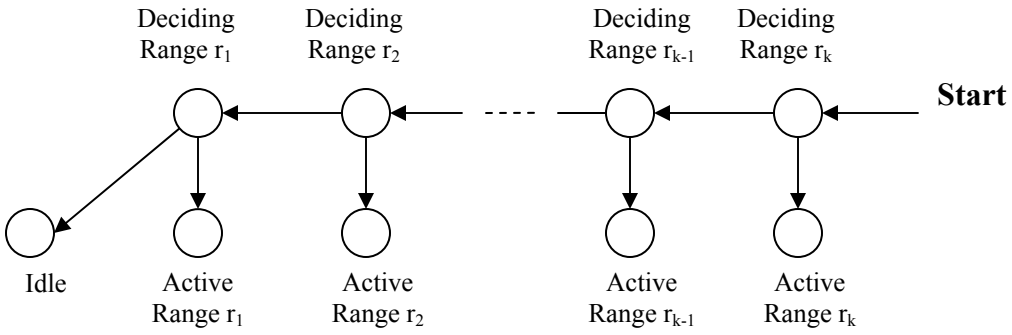


Fig. 4.2 State diagram of adjustable range sensor.

We first assume that each sensor *s* can communicate with its neighboring sensors within two times of maximum sensing range. In order to find sensor cover schedule, each sensor initially broadcasts its battery level and covered targets to all neighbors, and then stays in the deciding state with its maximum sensing range. Each sensor will change its state by the following transition rules:

When a sensor is in the deciding state with range *r*, then it should change its state into

- ***Active state*** with sensing range *r* if there is a target at range *r* which is not covered by any other active or deciding sensors.
- ***Deciding State***, but decreases its sensing range to the next furthest target if all covered targets at range r are covered by either active sensors or deciding sensors with a larger monitoring time.
- ***Idle state*** if when a sensor decreases its range to zero.

After all sensors decide their states to active or idle, each sensor will stay in that state for a certain period of time called, shuffle time, or until there is an active sensor which exhausts its energy supply and is going to die. All sensors are alerted using the wake-up call causing all sensors to change their state back to the deciding state with their maximum sensing range again. Finally, when there is a target which cannot be covered by any sensors, the network fails.

As in [5], [6], we can also show that ALBP satisfies the following properties:

Theorem 1: ALBP is a correct protocol. Each global reshuffle of ALBP needs 2 broadcasts (to the neighbors) from each sensors and the resultant set of all active sensors form a minimal sensor cover:

Proof: In ALBP, a sensor can change its states to IDLE only when its sensing range reaches to zero, i.e., any of its targets are covered by other active sensors. In other words, for any target t, there is a sensor with the largest battery to cover t. □

Theorem 2: The time complexity of ALBP is $O(\Delta^2)$ and the message complexity is $O(n\Delta)$ where Δ is the number of neighbors.

Proof: Let us investigate the time complexity for the worst case. For each shuffle time, each sensor receives a message, which contains targets and battery supply information, from one or more neighbors. However, a sensor node has no more than Δ neighbors. So a sensor can receive at most Δ messages, which also implies scanning over those in $O(\Delta)$ time. Moreover, in the worst case scenario, it may have to wait on all its neighbors to decide. Thus, the waiting time can accumulate as $O(\Delta + \Delta + \Delta + \ldots\ldots + \Delta)$ time for other neighbors to decide. Thus the time complexity is $O(\Delta^2)$.

Since each sensor has at most Δ neighbors and throughout the shuffling time, a sensor broadcasts at most two messages to its neighbors (first broadcast consists of its targets and battery, and the second broadcast is its status (on/off)), so each sensor sends at most $O(\Delta)$ messages in the decision phase. This means that the message complexity is $O(n\Delta)$, where n is the number of sensors. □

We will now show an example that ALBP can have an unbounded inefficiency as in LBP with the same example as in [6]. The network in Fig 4.3 consists of 3 targets, 2 sensors with

1000 battery each and two groups of 1000 sensors with 1-battery each. For this network, the optimal scheduling will be using the 1000 top right sensors and the bottom left 1000 group and next schedule will be the rest of sensors. Thus the total lifetime will be 2000.

However, ALBP suffers the same inefficiency as LBP. ALBP will use the two 1000 battery sensors until they are almost gone and the top target could not be the sensor cover after both the 1000-battery sensors die and the lifetime will be only 1000. It is easy to see that the factor 2 lost can be generalized as factor *k* loss [6].

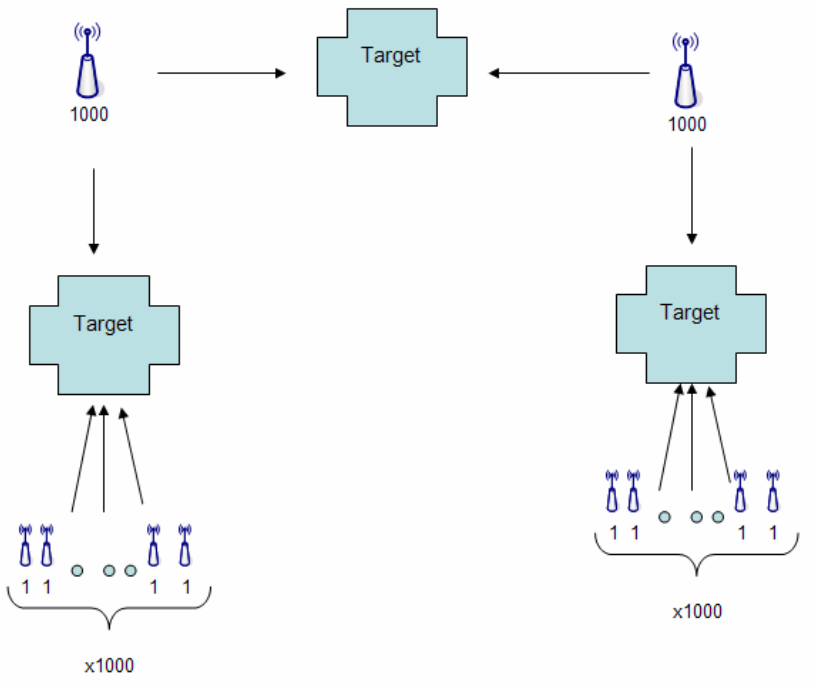


Figure 4.3 An example with the optimal schedule equal to 2000 time units long [6].

4.4 DEEPS Protocol for Adjustable Range Sensing (ADEEPS)

This section describes the Deterministic Energy-Efficient Protocol for Adjustable Range Sensing (ADEEPS).

At any moment, each sensor is in one of three states

- ***active***: the sensor monitors targets
- ***idle:*** the sensor listens to other sensors, but does not monitor targets
- ***deciding***: the sensor monitors targets, but will change its state to either active or idle state soon.

Before we define the transition rules, we have to decide which targets will be sinks and which will be hills, and also place at least one sensor in-charge of each target t. Here we use the maximum possible lifetime of the target, instead of total battery used in DEEPS for deciding sink and hill target. The lifetime of a sensor and the maximum lifetime of a target can be defined as follow: Let the lifetime of a sensor with battery b, with a given sensing range r ≤ maximum sensing range, and using energy mode e, be denoted by Lt (*b, r, e*). Then, the maximum lifetime of a target would be Lt (b_1, r_1, *e*) + Lt (b_2, r_2, *e*) + Lt (b_3, r_3, *e*) + …, assuming it can be covered by neighborhood sensors with batteries b_i at a distance r_i for i = 1, 2, …

Let ***sink*** be a target ***t*** which is poorest in maximum lifetime for at least one sensor covering ***t***. The abandoned target is a ***hill,*** i.e., a target which is not the poorest in maximum lifetime for any of its covering sensors. The following two rules determine which sensor should be in-charge of target ***t***:

1. If the target is a sink, then the sensors ***s*** covering ***t*** with the highest lifetime Lt (*b, r, e*) for which ***t*** is the poorest is placed in-charge of ***t***

2. If target ***t*** is a hill then the overall sensors covering ***t*** the sensor ***s*** whose poorest target has the largest lifetime is placed in-charge of ***t***. If there are several such sensors, then the richest among them is placed in-charge of ***t***.

The sensor's id is used to break the tie. We first assume that each sensor *s* can communicate with its neighboring sensors within four times of the maximum sensing range (same assumption as in [6]). In order to find the sensor cover schedule, each sensor initially broadcasts its corresponding lifetime and covered targets to all neighbors of neighbors, and then stays in the deciding state with its maximum sensing range. Each sensor will change its state by the following transition rules: When a sensor is in the deciding state with range r, then it should change its state into

- ***Active state*** with sensing range ***r,*** if there is a farthest target at range ***r*** less than or equal to ***r*** which is not covered by any other active or deciding sensors.
- ***Idle state***, whenever a sensor ***s*** is not in-charge of any target except those already covered by on-sensors, ***s*** switches itself to idle state.

After all sensors decide their state to active or idle, each sensor will stay in that state for a certain period of time (shuffle time) or until there is an active sensor which exhausts its energy supply and is going to die. All sensors are alerted using wake-up call causing all sensors to change their state back to deciding state with their maximum sensing range again. Finally, when there is a target which can not be covered by any sensors, the network fails.

Theorem 3: ADEEPS is a correct protocol. Each global reshuffle of DEEPS needs 2 broadcasts from each sensor and the resultant set of all active sensors form a minimal sensor cover.

Proof: The correctness of ADEEPS can be proved from the fact that each target has a sensor which is in-charge of that target and the transition rule to active state assures that the resultant sensor cover is minimal in which each sensor s has a target covered only by *s*.□

Theorem 4: The time complexity of ADEEPS is $O(\Delta^2)$ and the message complexity is $O(n\Delta^2)$ where Δ is the number of neighbors.

Proof: Let us investigate the time complexity for the worst case. For each shuffle time, each sensor receives a message, which contains target and battery supply information, from one or more neighbors and sensor node has no more than Δ neighbors. In ADEEPS, each sensor broadcast to its information to neighbors of neighbors. Thus, a sensor can receive at most Δ^2 messages. It needs $O(\Delta^2)$ time to run the ADEEPS algorithms as all the decisions regarding sink/hill targets, in-charge sensors, and active/idle can be taken locally (i.e., without waiting on neighboring sensors). Thus the time complexity is $O(\Delta^2)$.

Since each sensor has at most Δ neighbors and throughout the shuffling time, a sensor broadcasts at most two message to its neighbors of neighbors (first broadcast is its set of targets and its battery information, and second broadcast is its status (on/off)), so each sensor sends at most $O(\Delta^2)$ messages in the decision phase. This means that the message complexity is $O(n\Delta^2)$, where n is the number of sensors.□

ADEEPS does not suffer the inefficiency of LBP and ALBP. With reference to Fig4.3, the ADEEPS protocol will allow both 1000 sensors to be active simultaneously in the first shift. After that, the top target becomes the sinks and will be monitored only by one of the 1000-battery sensors.

5. SIMULATION OF THE ALGORITHMS

In this chapter we describe how the new algorithms are molded and how the simulation is carried out. We test the performance of the algorithm by simulating it over a wide range of simulation parameters. We start off this chapter by describing how the simulation is setup, implemented and operated. In the next chapter we present some results and analyze the results. Finally, we present the conclusions drawn from our simulation study.

5.1 Simulation Setup

To evaluate the performance of new algorithms and to make comparison with algorithms in [5], [6], [15], the new algorithms are implemented by using C++ in Windows XP operating system.

The simulator is designed to model a wide range of physical sensor network sizes with varying node densities. The location of the sensor nodes can be randomly deployed and the targets can also be placed randomly while creating the sensor and target inputs. For the simulation purpose, we created a static network of sensors scattered in a 100m x 100m area. The adjustable parameters are:

- N, the number of sensor nodes. We vary this from 40 to 200.
- M the number of targets. We vary this to 25 and 50.
- And the sensing range *r* which can vary smoothly from 5m to 30m/60m.

- The energy model can be either linear or quadratic energy as defined in [3]. The linear model defines the energy e_p needed to cover a target at distance r_p as $e_p = c_1 r_p$, where c_1 is constant. The quadratic model is defined as $e_p = c_2 r_p^{2,}$ where c_2 is a constant.

In order to make comparison, we used the same simulations parameters used in [15].

5.2 Explanation of Simulation

We implement the two basic algorithms LBP and DEEPS and they are further extended by using adjustable range sensing instead of fixed range sensing. The target and sensor files are generated using the parameters from section 5.1 and input into the program. We can vary the sensing range, and energy model from the command line and the lifetime of network is output as the result. For each algorithm, the following steps are required for the simulation:

1. Generate the target and sensor files which contain the information of the target id, target position, sensor id, sensor maximum battery, and sensor position.
2. Simulation is started from the command line wherein the target and sensor file, the maximum sensing range, and the energy model are provided as input.
3. Using these data and parameters, the simulation is started
4. The simulation runs until a target cannot be covered by sensors.
5. The simulations stops, and the lifetime of the network is printed out as the result.

5.3 Explanation of New Algorithms

We modify the distributed algorithms proposed in [5], [6] using the adjustable range sensors. In [5], [6], the authors proposed efficient distributed algorithms for improving network lifetime for fixed sensing range network.

The basic step in LBP and DEEPS is that each node has to decide whether they can go to sleep or become active and cover the targets. Each sensor knows its neighboring sensors and covered targets. After exchanging their battery power and covered targets, using the rules in chapter 4, each sensor decides whether they go to sleep or become active covering the target. In both algorithms, the decision is made only on the energy level, and does not consider the distance.

In the new algorithms ALBP and ADEEPS, both the energy level and distance are considered in the sensors' decisions. The following shows the steps in our simulation:

1. Targets and sensors are read into the memory.
2. Sensor nodes are in a deciding state and decide whether they can go to sleep or become active and cover the target.
3. Each sensor knows its neighboring sensors and covered targets.
4. For each sensor
    a. In ALBP, checks with each neighbor sensors starting from the farthest target whether that target can be covered by the neighbor sensor with larger battery level. If the neighbors target can cover the farthest target with larger battery level, then the sensor removes that target from the covered target list and reduces the sensing range to the next target. This

sensor will go to sleep if the range reaches zero. This process stops after all sensors make a decision.

b. In ADEEPS, each sensor decides which targets they are in-charge of by using the maximum lifetime of all the targets of its neighbors. After making this decision, each sensor decides to become active with range r ($r \leq$ maximum sensing range) or decides to sleep. This process stops after all sensors make a decision.

5. After all sensors decide their state to be active or idle, each sensor will stay in that state for a certain period of time (shuffle time) or until there is an active sensor which exhausts its energy supply and is going to die. All sensors are alerted using wake-up call causing all sensors to change their state back to the deciding state with their maximum sensing range and repeat the process from step 4.
6. This simulation is repeated until a target cannot be covered.
7. Then, the process terminates and the lifetime of the network is printed out.

6. SIMULATION RESULTS

In this section, we evaluate the performance of centralized and distributed algorithms and analyze the data generated from the simulations. We have simulated the four algorithms: LBP, ALBP, DEEPS and ADEEPS.

For the simulation environments, a static wireless network of sensors and targets which are scattered randomly in 100m x 100m area is considered. We assume that the communication range of each sensor is two times the sensing range. Simulations are carried out by varying the number of sensors and the lifetime is measured. We also vary the maximum sensing range, energy models, and numbers of targets with various combinations. The corresponding data and graphs are presented in the following sections.

In the first simulation, we compare the network lifetime computed by LBP, ALBP, DEEPS and ADEEPS by varying the number of sensors. In order to make a comparison with the distributed algorithm, AR-SC [3], we use the same parameters as theirs. The simulation is conducted with 25 randomly deployed targets, 40 to 200 sensors with an increment of 20. Each sensor has a maximum sensing range of 60m with linear energy model. The corresponding results are shown in table 6.1. The results from the simulations show that the lifetime increases with the increase in the number of sensor density because when more sensors are deployed, each target could be covered by more sensors.

Table 6.1 The lifetime of sensor networks with 25 targets.

Sensors	40	60	80	100	120	140	160	180	200
AR-SC [3]	20.0	25.0	31.0	44.0	49.0	53.0	62.0	68.0	75.0
LBP [5]	12.2	19.4	29.6	33.3	40.2	45.4	50.9	56.6	61.1
ALBP	15.0	20.4	28.6	35.3	45.7	56.8	56.7	62.2	68.3
DEEPS [6]	19.6	28.5	40.3	54.3	66.2	76.3	84.6	94.6	101.3
ADEEPS	24.6	35.6	49.6	68.4	83.4	92.7	105.9	118.6	124.7

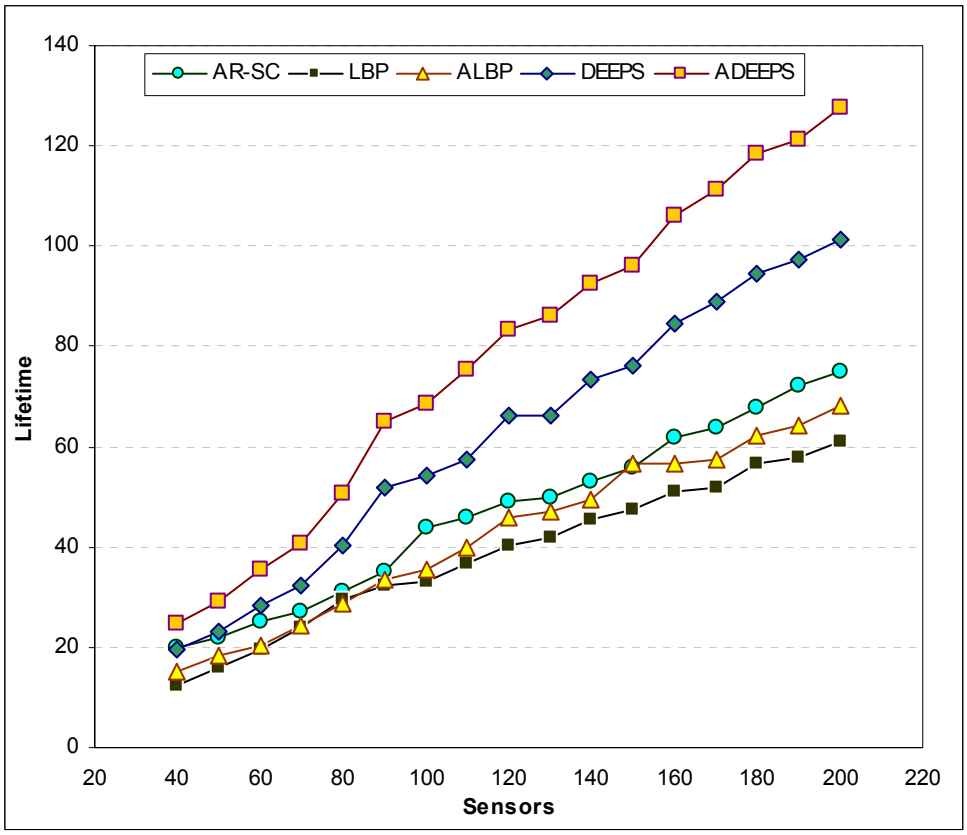


Figure 6.1 Variation in network lifetime with the number of sensors with 25 targets and linear energy model and 60m maximum sensing range.

In the second simulation, we vary the maximum sensing range to 30m. We use the same number of targets, sensors, and linear energy model. The results are consistent with the previous results because the network lifetime increases with the increase in the number of sensors. When compared to the result from Table 6.1/Fig6.1, adjusting the sensing ranges have an impact on network lifetime because when we decrease the sensing range, the network lifetime also decreases.

Table 6.2. The lifetime of sensor networks with 25 targets and 30m sensing range

Sensors	40	60	80	100	120	140	160	180	200
LBP [5]	5.48	10.51	11.22	12.51	15.12	16.75	16.75	18.37	23.97
ALBP	7.38	12.45	13.37	13.93	17.48	19.11	19.11	23.26	29.65
DEEPS [6]	7.22	14.13	16.22	18.01	22.69	26.03	26.03	27.66	33.43
ADEEPS	8.96	17.23	20.02	21.28	27.12	30.67	33.13	35.35	41.12

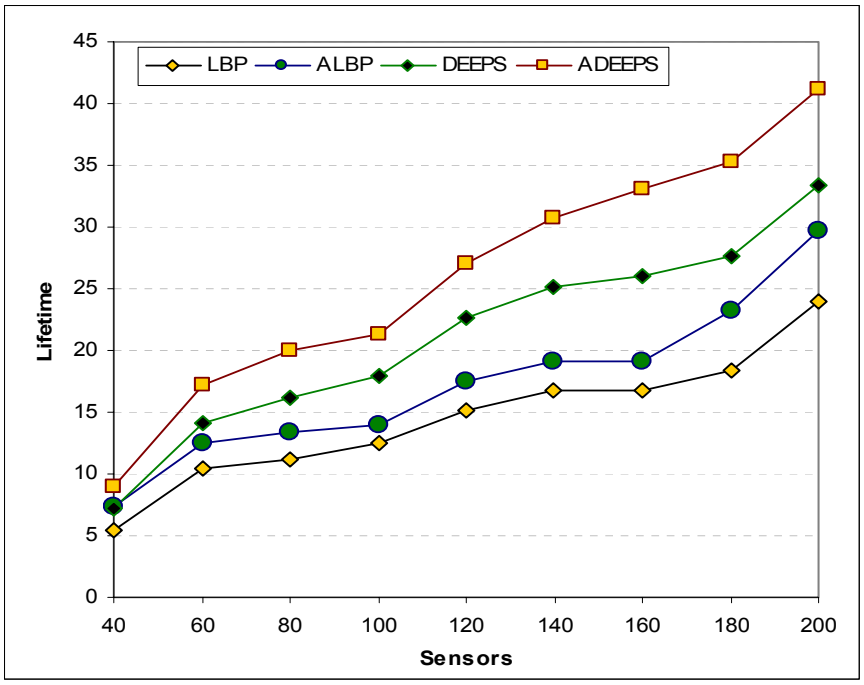


Figure 6.2. Variation in network lifetime with the number of sensors with 25 targets, linear energy model and 30m maximum sensing range.

In the third and the fourth simulation (Fig 6.3/6.4), we study the network lifetime while increasing the number of targets to 50 and vary the maximum sensing range to 30m and 60m. The numbers of sensors are varied from 40 to 200 with an increment of 20 and the energy model is linear. The results of simulations are consistent and showed that the network lifetime increases with the number of sensors. When compared with the results in experiments 1 and 2, the network lifetime decreases as more targets are monitored.

Table 6.3. The lifetime of sensor networks with 50 targets and 60m maximum sensing range

No. of Sensors	40	60	80	100	120	140	160	180	200
AR-SC [3]	-	18.6	24.3	30.2	39.6	48.3	54.2	60.1	65.78
LBP [5]	10.5	17.3	24.9	28.3	35.3	37.9	44.6	48.9	54.2
ALBP	11.7	18.1	26.2	30.3	38.0	40.8	47.5	51.9	58.1
DEEPS [6]	15.8	22.7	26.8	35.3	49.0	61.1	70.4	79.1	87.1
ADEEPS	18.0	28.2	33.7	38.9	56.8	75.9	90.1	98.6	108.3

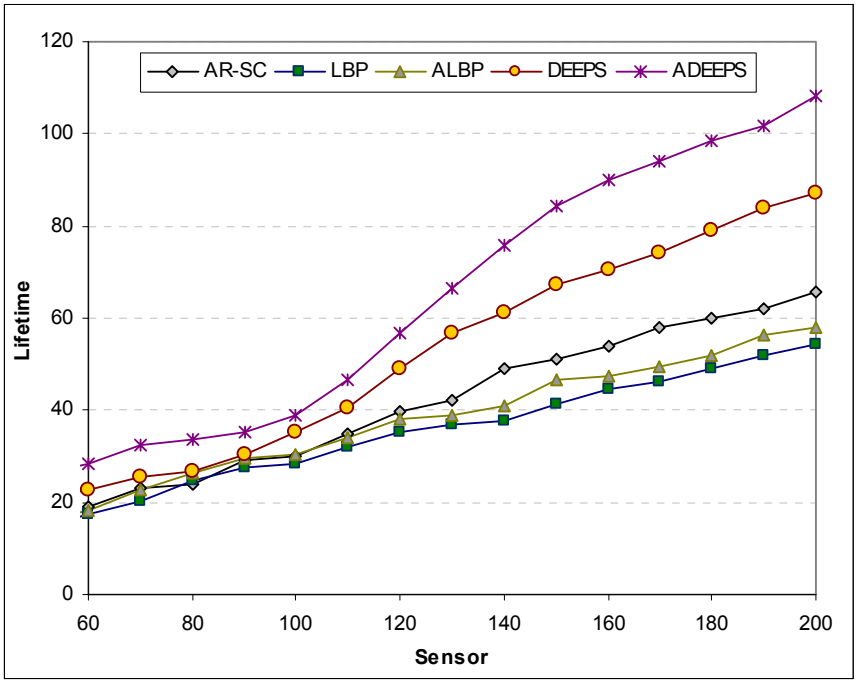


Figure 6.3 Variation in network lifetime with the number of sensors with 50 targets, linear energy model and 60m maximum sensing range

Table 6.4. The lifetime of sensor networks with 50 targets and 30m sensing range

No. of Sensors	40	60	80	100	120	140	160	180	200
LBP [5]	4.80	8.65	8.90	10.12	12.74	14.13	14.14	16.73	18.76
ALBP	5.38	9.42	9.77	11.84	14.31	15.39	15.39	18.03	19.99
DEEPS [6]	6.95	11.32	12.11	14.86	18.76	21.22	22.56	24.95	27.66
ADEEPS	8.17	13.45	14.91	17.55	23.20	26.40	28.20	30.10	32.89

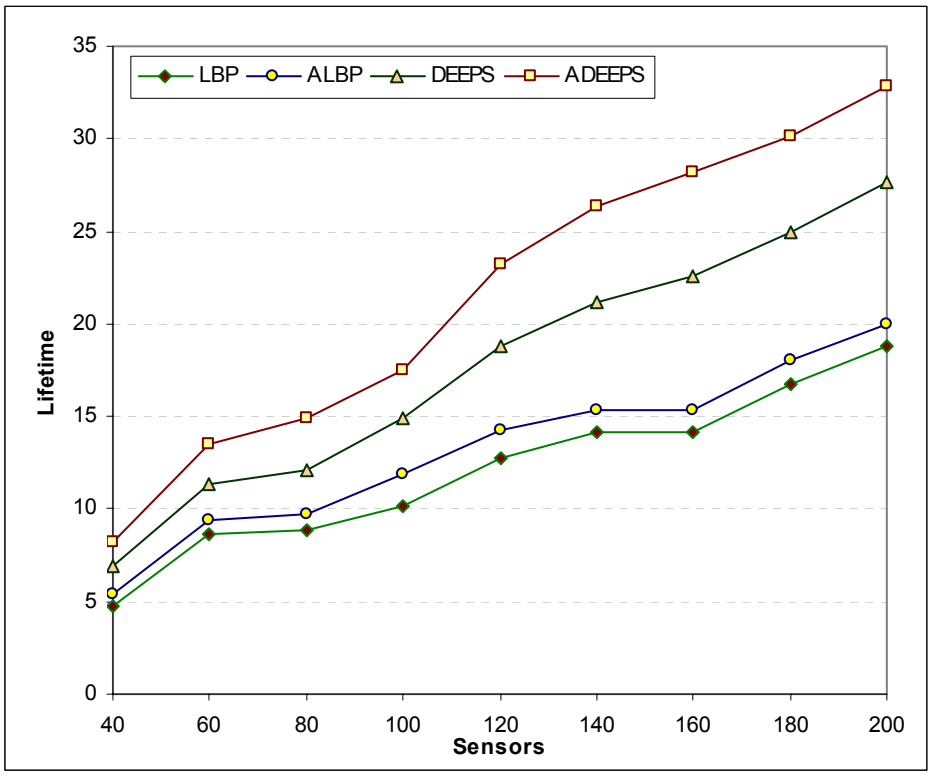


Figure 6.4 Variation in network lifetime with the number of sensors, with 50 targets, linear energy model and 30m maximum sensing range

In the fifth simulation (Table 6.5/Fig 6.5), we conduct with the quadratic energy model. We use the same number of sensors (40 to 200 with increment of 20), the maximum sensing range is 30m and the energy model is quadratic. For both energy models, the result indicates that the network lifetime increases with the number of sensors. Another interesting fact is that the network lifetime is significantly improved with ALBP and ADEEPS in the quadratic model. This phenomenon is quite logical since in the fixed sensing model, each sensor consumes more energy than the adjustable sensing range model.

Table 6.5. The lifetime of sensor networks with 25 targets and quadratic energy model

No. of Sensors	40	60	80	100	120	140	160	180	200
LBP [5]	1.90	3.85	4.25	4.75	5.78	6.41	6.41	7.35	8.14
ALBP	3.56	5.91	6.11	6.60	8.47	9.09	10.10	13.06	17.80
DEEPS [6]	3.80	7.70	8.49	9.51	11.57	12.83	12.83	14.69	16.28
ADEEPS	7.18	12.2	16.05	17.97	21.86	23.98	24.56	27.77	30.2

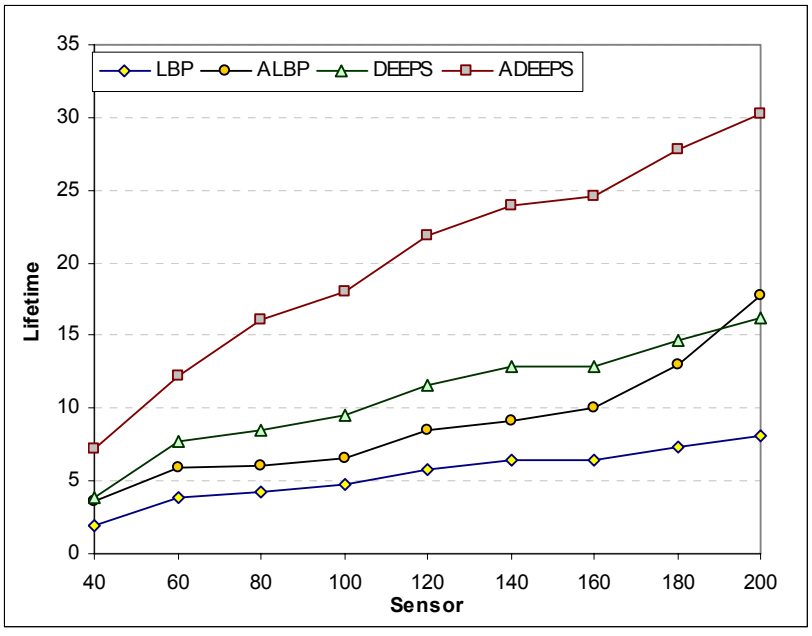


Figure 6.5 Variation in network lifetime with the number of sensors, with 25 targets, quadratic energy model and 30m maximum sensing range.

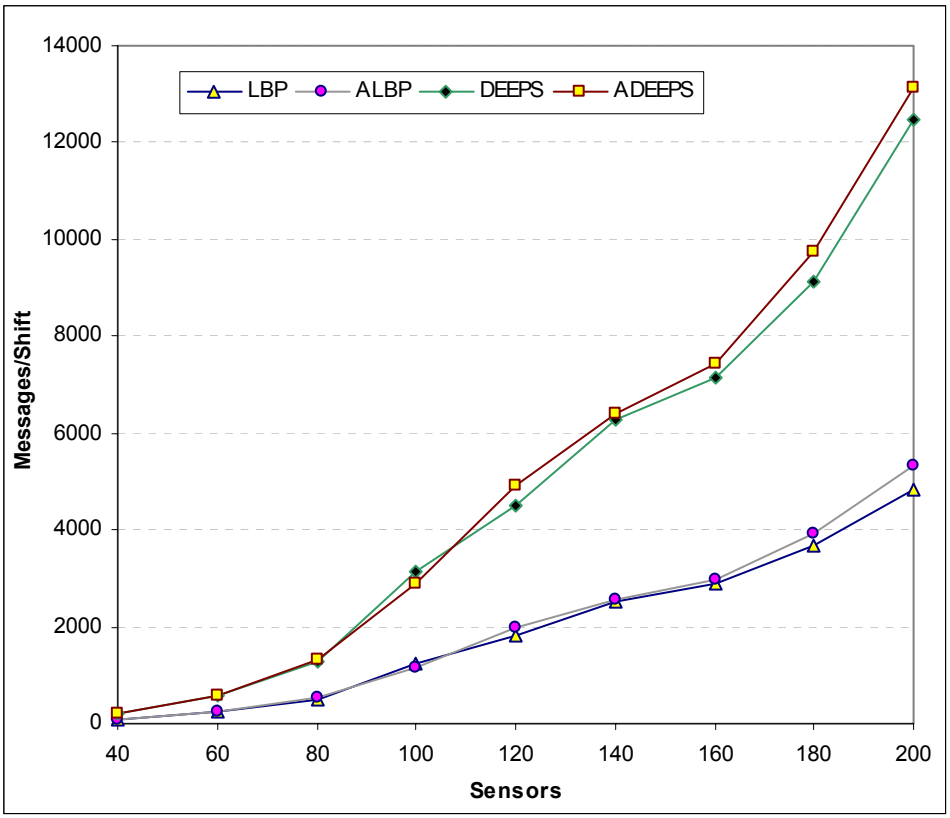


Fig 6.6. Average numbers of messages sent during each shift

In Figure 6.6, we provide the average numbers of messages sent during each shift. It can be seen that more messages are sent when the number of deployed sensors increases and the average messages sent in DEEPS and ADEEPS are much higher than LBP and ALBP. This is because in DEEPS and ADEEPS the communication range is four times higher than the sensing range and each sensor has more neighbors and needs to send more messages (in effect communicating with 2-hop neighbors).

The above tables and figures show the variation in network lifetime while varying the number of sensors, number of targets, maximum sensing ranges, and different energy models. From the results, the overall improvement in network lifetime of ALBP over LBP is around 10 percent and ADEEPS over DEEPS is about 20 percent for linear energy model. For quadratic energy model, the improvements are much more.

7. CONCLUSIONS & FUTURE WORK

In this paper, we provide a problem formulation for the lifetime maximization problem in a sensor network with adjustable sensing ranges. We then extended the two distributed algorithms proposed in [5], [6] with adjustable sensing ranges. We also provide the analysis to show the correctness and efficiency of ALBP and ADEEPS and demonstrate it using the simulation results. The simulation results verify that with the adjustable sensing range, the network lifetime can be improved. The simulations results can be summarized as follows:

- For the given number of targets and sensing ranges, the network lifetime increases with the number of sensors. When the number of targets is increased, the network lifetime decreases as more targets are monitored.
- Network lifetime increases with an increase in the sensing range.
- With adjustable sensing range, the network lifetime increases, and the increase is more dramatic with quadratic energy models.

The future work will include simulating these algorithms with the combination of communication protocols, and improving the performance of the distributed algorithms by reducing its overheads as well as better integration of adjustable sensing range into the algorithms.

BIBLIOGRAPHY

[1] Chee-Yee Chong and Srikanta P. Kumar, “Sensor Networks: Evolution, Opportunities and Challenges”. *Proceeding of the IEEE*, vol. 91, no. 8, Aug. 2003.

[2] R. Hahn and H. Reichl, “Batteries and power supplies for wearable and ubiquitous computing”, *in Proc. 3rd Intl. Symposium on Wearable computers*, 1999.

[3] M. Cardei, J. Wu, N. Lu, M.O. Pervaiz, “Maximum Network Lifetime with Adjustable Range”, *IEEE Intl. Conf. on Wireless and Mobile Computing, Networking and Communications (WiMob'05),* Aug. 2005.

[4] G. J. Pottie and W. J. Kaiser, “Wireless integrated network sensors”, *Communication ACM,* 43(5):51-58, 2000.

[5] P. Berman, G. Calinescu, C. Shah and A. Zelikovsky, "Power Efficient Monitoring Management in Sensor Networks," *IEEE Wireless Communication and Networking Conference (WCNC'04)*, pp. 2329-2334, Atlanta, March 2004.

[6] Brinza, D. and Zelikovsky, A, “DEEPS: Deterministic Energy-Efficient Protocol for Sensor networks”, *ACIS International Workshop on Self-Assembling Wireless Networks (SAWN'06), Proc. of SNPD, pp*. 261-266, 2006.

[7] M. Cardei, J. Wu, “Energy-Efficient Coverage Problems in Wireless Ad-Hoc Sensor Networks”, *Computer Communications Journal (Elsevier),* Vol.29, No.4, pp. 413-420, Feb. 2006.

[8] Jim Kurose and Keith Ross, “Computer Networking: A Top Down Approach Featuring the Internet”, *3rd edition. Addison-Wesley*, July 2004.

[9] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci, “A Survey on Sensor Networks”, *IEEE Communications Magazine*, pp 102-114, Aug. 2002.

[10] M. Cardei, M.T. Thai, Y. Li, and W. Wu, “Energy-efficient target coverage in wireless sensor networks”, I*n Proc. of IEEE Infocom*, 2005.

[11] V. Raghunathan, C. Schurgers, S. Park, and M. B. Srivastava, “Energy-Aware Wireless Microsensor Networks”, *IEEE Signal Processing Magazine*, 19 (2002), pp 40-50.

[12] T. Yan, T. He, and J. Stankovic, “Differentiated surveillance for sensor networks”, *In Proceedings of Sensys*, 2003.

[13] M. Cardei and D.-Z. Du, “Improving Wireless Sensor Network Lifetime through Power Aware Organization”, *ACM Wireless Networks*, vol. 11, No. 3, May 2005.

[14] P. Berman, G. Calinescu, C. Shah and A. Zelikovsky, "Efficient Energy Management in Sensor Networks," *In Ad Hoc and Sensor Networks, Wireless Networks and Mobile Computing, Volume 2* , Y. Xiao and Y. Pan (Eds.), Nova Science Publishers. 2005.

[15] A. Dhawan, C. T. Vu, A. Zelikovsky, Y. Li, and S. K. Prasad, “Maximum Lifetime of Sensor Networks with Adjustable Sensing Range”, *2nd ACIS International Workshop on Self-assembling Wireless Networks,* (SAWN 2006), Las Vegas, NV, June 19-20, 2006.

[16] J. Carle and D. Simplot, “Energy Efficient Area Monitoring by Sensor Networks”, *IEEE Computer*, Vol 37, No 2 (2004) 40-46.

[17] D. Tian and N. D. Georganas, “A Coverage-Preserving Node Scheduling Scheme for Large Wireless Sensor Networks”, *Proc. of the 1st ACM Workshop on Wireless Sensor Networks and Applications,* 2002.

[18] X. Wang, G. Xing, Y. Zhang, C. Lu, R. Pless, and C. D. Gill, "Integrated Coverage and Connectivity Configuration in Wireless Sensor Networks", *First ACM Conference on Embedded Networked Sensor Systems*, 2003.

[19] J. Wu and S. Yang, "Coverage and Connectivity in Sensor Networks with Adjustable Ranges", *International Workshop on Mobile and Wireless Networking (MWN)*, Aug. 2004.

[20] H. Zhang and J. C. Hou, "Maintaining Sensing Coverage and Connectivity in Large Sensor Networks", *NSF International Workshop on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks*, Feb. 2004.

[21] J. Wu and S. Yang, "Coverage and Connectivity in Sensor Networks with Adjustable Ranges", *International Workshop on Mobile and Wireless Networking (MWN)*, Aug. 2004.

[22] D. Brinza, G. Calinescu, S. Tongngam, and A. Zelikovsky, "Energy-Efficient Continuous and Event-Driven Monitoring", *In Proc. 2nd IEEE International Conference on Mobile Ad-Hoc and Sensor Systems*, 2005.

[23] L. Gu and J. Stankovic, "Radio triggered wake-up capability for sensor networks", *In Real-Time Applications Symposium,* May 2004.

[24] N. Garg and J. Könemann, "Faster and simpler algorithms for multi commodity flows and other fractional packing problems", *Proc. 39th Annual Symposium on the Foundations of Computer Science*, 1998, pp 300-309.

[25] R. Hahn and H. Reichl, "Batteries and power supplies for wearable and ubiquitous computing", *in Proc. 3rd Intl. Symposium on Wearable computers*, 1999

[26] J. Pottie and W. J. Kaiser, "Wireless integrated net-work sensors," *Communication ACM*, 43(5):51- 58, 2000.

[27] Photo Electric Sensors,

http://www.schneider-electric.ca/www/en/products/sensors2000/html/abc.htm

APPENDIX - A

C++ code for ALBP

```
#include <iostream>
#include <vector>
#include <string>
#include <math.h>
#include <fstream>

using namespace std;

class target{
        protected:
                int id;
                float x;
                float y;
        public:
                target(){};
                target(int i, float xpos, float ypos){
                        id=i; x=xpos; y=ypos;};
                double distance(double x1, double y1);
                float getx(){ return x;};
                float gety(){ return y;};
                int getid(){ return id;};

};
double target::distance(double x1, double y1){
        return sqrt((x1-x)*(x1-x)+(y1-y)*(y1-y));
}
class sensor {
        protected:
                int id;
                float status;
                float x,y;
                float battery;
                double maxRange;
                double range;
                bool linearPower;
```

```
public:
        vector <target> coveredTarget;
        vector <sensor*> neighbor;

        // constructor
        sensor (int ID,float xpos, float ypos,float maxb, double maxr, bool linear);


        // function to find distance
        double distance(double x1, double y1);

        // function to find target
        void getTarget( vector<target> t);

        // function to find neighbor sensor
        void getNeighbor(vector<sensor> &s);

        // find time with specific range
        double getTime(){
                if (linearPower)
                        return (battery/range);
                else
                        return (battery/(range*range));};

        // decrease range
        void decreaseRange();

        //active with range k
        void goactive(vector<target> t){status = range;getTarget(t);};

        // go sleep
        void gosleep() {status =0; range = 0;};

        // get status
        float getstatus(){return status;};

        // set new battery
        void setBattery(double time){
                if (linearPower)
                        battery -= range*time;
                else
                        battery -= range*range*time;};

        // wakeup
        void wakeup(){status = -1; range =maxRange;};
```

```
		// is sensor dead
		bool isDead(){ if (battery == 0) return true; else return false;};

		// can sensor cover a target with maximum range
		bool canCoveredTarget(target &t);

		// is sensor cover a target
		bool covers(target &t);

		// get sensor id
		int getId(){return id;};

		// get Battery level;
		int getBatteryLevel(){ return battery;};

};

bool sensor::canCoveredTarget(target &t){
	if (t.distance(x,y) <= maxRange)
		return true;
	else
		return false;
}


bool sensor::covers(target &t){
	if (t.distance(x,y) <= range)
		return true;
	else
		return false;
}


void sensor::decreaseRange(){
	coveredTarget.erase(coveredTarget.begin());
	if (coveredTarget.empty()){
		range = 0;
		gosleep();
	}
	else
		range = coveredTarget[0].distance(x,y);
}

double sensor::distance(double x1, double y1){
	return sqrt((x1-x)*(x1-x)+(y1-y)*(y1-y));
```

```
}

sensor::sensor(int ID, float xpos, float ypos, float maxb, double maxr, bool linear){
	id = ID; x = xpos; y = ypos; battery = maxb; maxRange = maxr;linearPower = linear;
}

void sensor::getTarget(vector<target> t){
	double distance;
	bool alreadyadd;
	vector<target>::iterator itr;

	coveredTarget.clear();

	for(int i=0;i < t.size(); i++){
		distance = t[i].distance(x,y);
		if (distance <= range){
			alreadyadd = false;
			for (itr = coveredTarget.begin(); itr < coveredTarget.end(); itr++){
				if (itr->distance(x,y) < distance){
					coveredTarget.insert(itr,t[i]);
					alreadyadd = true;
					break;
				}
			}
			if (!alreadyadd)
				coveredTarget.push_back(t[i]);

		}
	}

	if (!coveredTarget.empty())
		range = coveredTarget[0].distance(x,y);
	else
		range = 0;

}

void sensor::getNeighbor(vector<sensor> &s){
	double distance;
	neighbor.clear();

	for(int i=0;i < s.size(); i++){
		// find distance
		distance = s[i].distance(x,y);

		// add to vector neighbor if a sensor is in 2*maxRange
```

```
                if ((distance <= 2*maxRange) && (distance != 0)){
                        neighbor.push_back(&s[i]);

                }
        }
}

void getInformation (vector<target> &t, vector<sensor> &s);
bool isAllDecided (vector<sensor> s);
void deleteSensor (vector<sensor> &s);
bool allTargetcovered (vector<target> t, vector<sensor> s);

int main()
{
        vector<target> t;
        vector<sensor> s;
        target T;
        double batteryTime, nbatteryTime ;
        int targetid;
        bool coveredbyother = false;
        bool decreaseRange = false;
        double shuffleTime = 1;
        int             shift = 1; // schuffle index
        double shiftTime;       // time for one shift
        double monitorTime = 0;       // total monitor time

        // get target and sensor information
        getInformation(t, s);

        while(allTargetcovered(t,s)){

                // for each sensor, find target and neighbor sensor
                for(int i=0;i < s.size(); i++){
                        s[i].wakeup();
                        // get target
                        s[i].getTarget(t);
                        // find neighbor sensor
                        s[i].getNeighbor(s);
                }

                // start by assume that next shift time = shuffle time
                shiftTime = shuffleTime;

                // shuffle process
                while (!isAllDecided(s)){
```

```
// for every sensor
for(int i=0;i < s.size(); i++){
   if (s[i].getstatus() < 0){
         decreaseRange = false;
         if (!s[i].coveredTarget.empty()){
                  batteryTime = s[i].getTime();
                  // check the first target: the one with maximum length
                  targetid = s[i].coveredTarget[0].getid();
                           coveredbyother = false;
                           // check with all neighbors
                           for (int j=0; j< s[i].neighbor.size(); j++){
                           // every covered target of neightbor j
                           for (int k=0; k< s[i].neighbor[j]->coveredTarget.size(); k++){
                           // the target can be covered by neighbor j
                           if (targetid == s[i].neighbor[j]->coveredTarget[k].getid()){
                              coveredbyother = true;
                              nbatteryTime = s[i].neighbor[j]->getTime();
                               i f (batteryTime < nbatteryTime || (nbatteryTime ==
                           batteryTime && s[i].getId()<s[i].neighbor[j]->getId())
                                    || s[i].neighbor[j]->getstatus()>0){
                                       s[i].decreaseRange();
                                       decreaseRange = true;
                                      break;
                                    }
                           }
                  }
                  if (decreaseRange){
                    break;
                   }
         }
          if (!coveredbyother){
         s[i].goactive(t);
         if (shiftTime > batteryTime){
                  shiftTime = batteryTime;
                                    }

         // notify with all neighbors
         for (int j=0; j< s[i].neighbor.size(); j++){
         // every covered target of neightbor j
         while(!s[i].neighbor[j]->coveredTarget.empty()){
                  T = s[i].neighbor[j]->coveredTarget[0];
                  if (s[i].covers(T)){
                    s[i].neighbor[j]->decreaseRange();

                  } else
```

```
                    // not covered
                    break;
                        }
                                                    }
                                                }
                                    } else{
                                                s[i].gosleep();
                                    }
                                }
                        }
                }


        // end of each shift
        monitorTime += shiftTime;
        for (int i=0; i< s.size(); i++)
         s[i].setBattery(shiftTime);

        // print each shift detail
        cout << "Shift #" << shift << ", shift time =" << shiftTime << " hr.  Life time = " < <
        monitorTime << " hr." << endl ;
                for (int i=0; i< s.size(); i++){
                        cout << " - " << s[i].getId() << ": Battery :"<< s[i].getBatteryLevel();
                        cout << "\tStatus = ";
                        if (s[i].getstatus() == 0)
                                cout << "Sleep"<< endl;
                        else
                                cout << "Active"<< " with range = " << s[i].getstatus() << endl;
                }
                shift++;

                // delete dead sensor from vector sensor
                deleteSensor(s);

                cout << endl << endl << endl;
        }

        cout << "Sensor life : " << monitorTime << endl<< endl;
        return 0;

}

void getInformation (vector<target> &t, vector<sensor> &s){

        float xpos, ypos, maxb, maxr;
        string filename;
```

```
int tid, sid, choice, numSensor,numTarget;
bool linearPower;

// read target information
cout << "Enter target file:";
getline(cin,filename);
ifstream targetfile (filename.c_str());

if (targetfile.is_open()){
        targetfile >> numTarget;
        cout << "numTarget :" << numTarget << endl;
        for(int i =1; i<=numTarget; i++){
                targetfile >> tid;
                targetfile >> xpos;
                targetfile >> ypos;
                // add new target to vector
                t.push_back(target(tid,xpos,ypos));
        }
        targetfile.close();
}else {
        cout << "can not open file " << filename << endl;
}

// read sensor information
cout << "Enter sensor file :";
getline(cin,filename);
ifstream sensorfile (filename.c_str());

cout << "Maximum Range :";
cin >> maxr;

// power function
cout << "Power function, 1. Linear 2.Quadratic :";
cin >> choice;
if (choice == 1)
        linearPower = true;
else
        linearPower = false;

if (sensorfile.is_open()){
        sensorfile >> numSensor;
        cout << "numSensor :" << numSensor << endl;
        for(int i=1;i<=numSensor; i++){
                sensorfile >> sid;
                sensorfile >> maxb;
                sensorfile >> xpos;
```

```
                sensorfile >> ypos;
                // add the new sensor to vector
                s.push_back(sensor( sid, xpos, ypos, maxb, maxr, linearPower));
        }
        sensorfile.close();
    }else {
        cout << "can not open file " << filename << endl;
    }
}

bool isAllDecided (vector<sensor> s){
    for (int i=0; i< s.size(); i++){
        if (s[i].getstatus() < 0)
            return false;
    }
    return true;

}

void deleteSensor (vector<sensor> &s){
    int numDead = 0;
    vector<sensor>::iterator itr;

    // find number of dead sensor
    for (itr = s.begin(); itr!= s.end(); itr++){
        if (itr->isDead()){
            numDead++;
        }
    }

    // delete sensor
    for (int i=1; i<= numDead; i++){
        for (itr = s.begin(); itr!= s.end(); itr++){
            if (itr->isDead()){
                cout << "sensor: " << itr->getId() << " dead " << endl;
                s.erase(itr);
                break;
            }
        }
    }
}

bool allTargetcovered (vector<target> t, vector<sensor> s){
    bool tcovered;

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

```
            bool tcovered = false;
            for (int j=0; j<s.size(); j++){
                if (s[j].canCoveredTarget(t[i])){
                    tcovered = true;
                    break;
                }
            }
            if (!tcovered){
                cout << "target : " << t[i].getid() << "is not covered"<< endl;
                return false;
            }
        }

        return true;
}
```

APPENDIX - B

C++ code for ADEEPS

```
#include <iostream>
#include <vector>
#include <string>
#include <math.h>
#include <fstream>
using namespace std;

class target{
        protected:
                int id;
                float x;
                float y;
                bool sink; //true if target is a sink for any covering sensor
                float totalBat;
        public:
                target(){};
                target(int i, float xpos, float ypos){
                        id=i; x=xpos; y=ypos;};
                double distance(double x1, double y1);
                float getx(){ return x;};
                float gety(){ return y;};
                int getid(){ return id;};
                bool isSink(){ return sink;}; //returns true if the target is a sink, false otherwise
      void setSink(bool val) { sink=val; };
      void setTotalBat(float inBat){ totalBat=inBat; };
      float getTotalBat () { return totalBat; };
};

double target::distance(double x1, double y1){
        return sqrt((x1-x)*(x1-x)+(y1-y)*(y1-y));
}

class sensor {
        protected:
                int id;
                float status;
                float x,y;
                float battery;
                double maxRange;
                double range;
                bool linearPower;
```

```
public:
        vector <target> coveredTarget;
        vector <sensor*> neighbor;

        // constructor
        sensor (int ID,float xpos, float ypos,float maxb, double maxr, bool linear);


        // function to find distance
        double distance(double x1, double y1);

        // function to find target
        void getTarget( vector<target> t);

        // function to find neighbor sensor
        void getNeighbor(vector<sensor> &s);

        // find time with specific range
        double getTime(){
                if (linearPower)
                        return (battery/range);
                else
                        return (battery/(range*range));};

        // decrease range
        void decreaseRange();

        //active with range k
        void goactive(vector<target> t){status = range;getTarget(t);};

        // go sleep
        void gosleep() {status =0; range = 0;};

        // get status
        float getstatus(){return status;};

        // set new battery
        void setBattery(double time){
                if (linearPower)
                        battery -= range*time;
                else
                        battery -= range*range*time;};

        // wakeup
        void wakeup(){status = -1; range =maxRange;};
```

```
		// is sensor dead
		bool isDead(){ if (battery == 0) return true; else return false;};

		// can sensor cover a target with maximum range
		bool canCoveredTarget(target &t);

		// is sensor cover a target
		bool covers(target &t);

		// get sensor id
		int getId(){return id;};

		// get Battery level;
		int getBatteryLevel(){ return battery;};

		//Go active with maximum range
		void goactiveMaxrange(vector<target> t){status = maxRange;getTarget(t);};

		// Removed a target from Sensor's covered target list
		void removedCoveredtarget();

};

bool sensor::canCoveredTarget(target &t){
	if (t.distance(x,y) <= maxRange)
		return true;
	else
		return false;
}


bool sensor::covers(target &t){
	if (t.distance(x,y) <= range)
		return true;
	else
		return false;
}


void sensor::decreaseRange(){
	coveredTarget.erase(coveredTarget.begin());
	if (coveredTarget.empty()){
		range = 0;
		gosleep();
	}
	else
```

```
		range = coveredTarget[0].distance(x,y);
}

double sensor::distance(double x1, double y1){
	return sqrt((x1-x)*(x1-x)+(y1-y)*(y1-y));
}

sensor::sensor(int ID, float xpos, float ypos, float maxb, double maxr, bool linear){
	id = ID; x = xpos; y = ypos; battery = maxb; maxRange = maxr;linearPower = linear;
}

void sensor::getTarget(vector<target> t){
	double distance;
	bool alreadyadd;
	vector<target>::iterator itr;

	coveredTarget.clear();

	for(int i=0;i < t.size(); i++){
		distance = t[i].distance(x,y);
		if (distance <= range){
			alreadyadd = false;
			for (itr = coveredTarget.begin(); itr < coveredTarget.end(); itr++){
				if (itr->distance(x,y) < distance){
					coveredTarget.insert(itr,t[i]);
					alreadyadd = true;
					break;
				}
			}
			if (!alreadyadd)
				coveredTarget.push_back(t[i]);

		}
	}

	if (!coveredTarget.empty())
		range = coveredTarget[0].distance(x,y);
	else
		range = 0;

}
```

```
void sensor::getNeighbor(vector<sensor> &s){
        double distance;
        neighbor.clear();

        for(int i=0;i < s.size(); i++){
                // find distance
                distance = s[i].distance(x,y);

                // add to vector neighbor if a sensor is in 2*maxRange
                if ((distance <= 4*maxRange) && (distance != 0)){
                        neighbor.push_back(&s[i]);

                }
        }
}

void sensor::removedCoveredtarget(){
        if (coveredTarget.empty()){
                range = 0;
                gosleep();
        }
   else

      coveredTarget.erase(coveredTarget.begin());


}

void getInformation (vector<target> &t, vector<sensor> &s);
bool isAllDecided (vector<sensor> s);
void deleteSensor (vector<sensor> &s);
bool allTargetcovered (vector<target> t, vector<sensor> s);
void setTarget (vector<target> t, vector<sensor> s);

int main()
{
        vector<target> t;
        vector<sensor> s;
        target T;
        double batteryTime, nbatteryTime ;
        int targetid;
        bool coveredbyother = false;
        bool decreaseRange = false;
        bool coverHill = false;
        double shuffleTime = 1;
        int shift = 1; // schuffle index
```

```
    double shiftTime;       // time for one shift
    double monitorTime = 0;       // total monitor time

    // get target and sensor information
    getInformation(t, s);

    while(allTargetcovered(t,s)){

 // calculate target total battery and set sinks
// this can be decide locally using the information form neighbors

setTarget(t,s);

// for each sensor, find target and neighbor sensor
            for(int i=0;i < s.size(); i++){
                    s[i].wakeup();
                    // get target
                    s[i].getTarget(t);
                    // find neighbor sensor
                    s[i].getNeighbor(s);
            }

            // start by assume that next shift time = shuffle time
            shiftTime = shuffleTime;

            // shuffle process
            while (!isAllDecided(s)){

    // for every sensor
                    for(int i=0;i < s.size(); i++){
                            if (s[i].getstatus() < 0){
                                    decreaseRange = false;
                                    if (!s[i].coveredTarget.empty()){
                                            batteryTime = s[i].getTime();

            //check whether sensor cover hills
            for (int m=0; m<s[i].coveredTarget.size(); m++) {
               if (!s[i].coveredTarget[m].isSink())
                  coverHill = true;
               break;

               }

            // check the first target: the one with maximum length
            targetid = s[i].coveredTarget[0].getid();
```

```
        if (s[i].coveredTarget[0].isSink()) {
          coveredbyother = false;
                // check with all neighbors
                  for (int j=0; j< s[i].neighbor.size(); j++){
        // every covered target of neightbor j
        for (int k=0; k< s[i].neighbor[j]->coveredTarget.size(); k++){
        // the target can be covered by neighbor j
         if (targetid == s[i].neighbor[j]->coveredTarget[k].getid()){
          coveredbyother = true;
           nbatteryTime = s[i].neighbor[j]->getTime();
         if ((batteryTime < nbatteryTime || (nbatteryTime == batteryTime &&
        s[i].getId()<s[i].neighbor[j]->getId())|| s[i].neighbor[j]->getstatus()>0) &&
        (!coverHill)){
                            s[i].decreaseRange();
                            decreaseRange = true;
                            break;
                             }
                    }
        }
        if (decreaseRange){
                             break;
          }
    }
}
        else {
            coveredbyother = false;
                // check with all neighbors
                for (int j=0; j< s[i].neighbor.size(); j++){
                // every covered target of neightbor j
                for (int k=0; k< s[i].neighbor[j]->coveredTarget.size(); k++){
                // the target can be covered by neighbor j
                if (targetid == s[i].neighbor[j]->coveredTarget[k].getid()){
                        coveredbyother = true;
                        nbatteryTime = s[i].neighbor[j]->getTime();
                        if (batteryTime < nbatteryTime || (nbatteryTime == batteryTime
                        && s[i].getId()<s[i].neighbor[j]->getId()) || s[i].neighbor[j]-
                        >getstatus()>0){
                                        s[i].decreaseRange();
                                        decreaseRange = true;
                                         break;
                                }
                        }
                }
                 if (decreaseRange){
                                        break;
            }
```

```
            }
          }


        if (!coveredbyother){
                s[i].goactive(t);
                if (shiftTime > batteryTime){
                shiftTime = batteryTime;
                }

        // notify with all neighbors
        for (int j=0; j< s[i].neighbor.size(); j++){
        // every covered target of neightbor j
        while(!s[i].neighbor[j]->coveredTarget.empty()){
        T = s[i].neighbor[j]->coveredTarget[0];
        if (s[i].covers(T)){
        s[i].neighbor[j]->decreaseRange();

        } else
        // not covered
        break;
            }
          }
        }
        } else{
                s[i].gosleep();
              }
        }
    }
}


// end of each shift
monitorTime += shiftTime;
for (int i=0; i< s.size(); i++)
s[i].setBattery(shiftTime);

// print each shift detail
cout << "Shift #" << shift << ", shift time =" << shiftTime << " hr.  Life time = " <<
monitorTime << " hr." << endl ;
for (int i=0; i< s.size(); i++){
                cout << " - " << s[i].getId() << ": Battery :"<< s[i].getBatteryLevel();
                cout << "\tStatus = ";
                if (s[i].getstatus() == 0)
                        cout << "Sleep"<< endl;
```

```
                else
                        cout << "Active"<< " with range = " << s[i].getstatus() << endl;
            }
            shift++;

            // delete dead sensor from vector sensor
            deleteSensor(s);

            cout << endl << endl << endl;
        }

        cout << "Sensor life : " << monitorTime << endl<< endl;
        return 0;

}

void getInformation (vector<target> &t, vector<sensor> &s){

        float xpos, ypos, maxb, maxr;
        string filename;
        int tid, sid, choice, numSensor,numTarget;
        bool linearPower;

        // read target information
        cout << "Enter target file:";
        getline(cin,filename);
        ifstream targetfile (filename.c_str());

        if (targetfile.is_open()){
            targetfile >> numTarget;
            cout << "numTarget :" << numTarget << endl;
            for(int i =1; i<=numTarget; i++){
                targetfile >> tid;
                targetfile >> xpos;
                targetfile >> ypos;
                // add new target to vector
                t.push_back(target(tid,xpos,ypos));
            }
            targetfile.close();
        }else {
            cout << "can not open file " << filename << endl;
        }

        // read sensor information
        cout << "Enter sensor file :";
        getline(cin,filename);
```

```
	ifstream sensorfile (filename.c_str());

	cout << "Maximum Range :";
	cin >> maxr;

	// power function
	cout << "Power function, 1. Linear 2.Quadratic :";
	cin >> choice;
	if (choice == 1)
		linearPower = true;
	else
		linearPower = false;

	if (sensorfile.is_open()){
		sensorfile >> numSensor;
		cout << "numSensor :" << numSensor << endl;
		for(int i=1;i<=numSensor; i++){
			sensorfile >> sid;
			sensorfile >> maxb;
			sensorfile >> xpos;
			sensorfile >> ypos;
			// add the new sensor to vector
			s.push_back(sensor( sid, xpos, ypos, maxb, maxr, linearPower));
		}
		sensorfile.close();
	}else {
		cout << "can not open file " << filename << endl;
	}
}

bool isAllDecided (vector<sensor> s){
	for (int i=0; i< s.size(); i++){
		if (s[i].getstatus() < 0)
			return false;
	}
	return true;

}

void deleteSensor (vector<sensor> &s){
	int numDead = 0;
	vector<sensor>::iterator itr;

	// find number of dead sensor
	for (itr = s.begin(); itr!= s.end(); itr++){
		if (itr->isDead()){
```

```
                numDead++;
            }
        }

        // delete sensor
        for (int i=1; i<= numDead; i++){
            for (itr = s.begin(); itr!= s.end(); itr++){
                if (itr->isDead()){
                    cout << "sensor: " << itr->getId() << " dead " << endl;
                    s.erase(itr);
                    break;
                }
            }
        }
}

bool allTargetcovered (vector<target> t, vector<sensor> s){
        bool tcovered;

        for (int i=0; i<t.size(); i++){
            bool tcovered = false;
            for (int j=0; j<s.size(); j++){
                if (s[j].canCoveredTarget(t[i])){
                    tcovered = true;
                    break;
                }
            }
            if (!tcovered){
                cout << "target : " << t[i].getid() << "is not covered"<< endl;
                return false;
            }
        }

        return true;
}

void setTarget (vector<target> t, vector<sensor> s){
    float inBat=0;
    int id=0;
    bool b=true;

    //calculate the total LifeTime for each targets
    for (int i=0; i<t.size(); i++) {
       for ( int j=0; j<s.size(); j++) {
            if (s[j].canCoveredTarget(t[i])) {
            inBat += s[j].getTime();
```

```
            }
        }
    t[i].setTotalBat(inBat);
    }

    //set the targets whether sinks or not
    for (int k=0; k<s.size(); k++) {
        s[k].getTarget(t);
       for (int l=0; l<s[k].coveredTarget.size(); l++) {
          if ( s[l].coveredTarget[l].getTotalBat() < s[l].coveredTarget[l+1].getTotalBat())
             id = s[l].coveredTarget[l].getid();
           else id = s[l].coveredTarget[l+1].getid();
        }
        t[id].setSink (b);
    }

}
```

APPENDIX - C

1. Sample targets File: 50 targets

The first line shows the total number of targets and each successive line includes the target's id, target's x-coordinate, and target's y-coordinate of each targets.

```
50
0       579     42
1       308     773
2       470     746
3       230     281
4       376     2
5       737     31
6       127     705
7       104     95
8       164     12
9       123     679
10      678     275
11      668     677
12      754     580
13      400     628
14      545     159
15      647     246
16      258     554
17      640     259
18      281     389
19      262     225
20      456     449
21      32      533
22      621     268
23      744     345
24      716     176
25      153     238
26      687     638
27      111     628
28      683     356
29      42      791
30      709     710
31      179     12
32      53      204
33      418     541
34      713     409
35      254     769
```

36	589	572
37	551	432
38	138	10
39	17	341
40	556	523
41	306	133
42	713	84
43	798	527
44	767	502
45	240	20
46	669	756
47	780	377
48	228	26
49	771	571

2. Sample sensors file: 100 sensors

The first line shows the total number of sensors and each successive line includes the sensor's id, sensor's total battery, sensor's x-coordinate, and then sensor's y-coordinate.

100

0	590	527	201
1	373	422	586
2	528	473	449
3	236	551	316
4	373	478	421
5	100	312	202
6	229	219	550
7	864	626	74
8	619	542	239
9	381	435	592
10	714	457	128
11	544	610	295
12	261	80	727
13	410	17	773
14	193	418	421
15	966	51	206
16	481	550	145
17	309	644	606
18	680	653	107
19	607	461	88
20	628	741	608
21	428	555	240
22	314	794	245
23	864	622	218

24	355	574	772
25	980	761	467
26	115	90	448
27	515	408	220
28	736	170	642
29	228	610	570
30	696	795	333
31	603	253	347
32	287	261	722
33	104	743	82
34	341	169	176
35	967	137	322
36	857	780	110
37	487	643	794
38	322	290	474
39	173	432	133
40	672	122	568
41	151	342	130
42	583	408	708
43	647	537	31
44	915	383	460
45	567	593	199
46	727	749	124
47	778	139	237
48	603	674	424
49	101	226	366
50	340	872	9
51	619	999	999
52	456	361	881
53	102	924	155
54	389	557	889
55	621	813	1
56	308	763	846
57	988	313	315
58	431	394	194
59	414	885	186
60	829	568	777
61	689	643	952
62	115	964	428
63	630	672	337
64	641	874	454
65	228	636	539
66	572	307	325
67	257	499	373
68	428	534	383
69	56	878	795

70	747	471	418
71	619	128	940
72	66	223	180
73	796	708	532
74	940	152	512
75	977	643	817
76	945	767	818
77	967	261	589
78	217	724	160
79	500	923	424
80	493	962	733
81	304	404	17
82	314	170	392
83	120	212	566
84	704	364	776
85	458	175	458
86	374	844	808
87	719	992	197
88	853	213	598
89	670	360	245
90	231	909	384
91	758	61	994
92	22	349	163
93	127	70	814
94	634	718	487
95	714	911	967
96	430	453	527
97	559	594	847
98	881	672	942
99	302	43	486
100	989	113	364