2007

GPSense: an algorithmic framework for intelligent sensing at node level in WSN

Soujanya Soni

GPSense: an algorithmic framework for intelligent sensing at node level in WSN

by

Soujanya Soni

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science

Rochester Institute of Technology

2007

Approved by

Rajendra K. Raj
Dr Rajendra K. Raj, Chairperson

Leon Reznik
Dr Leon Reznik, Reader

Hans-Peter Bischof
Dr Hans-Peter Bischof, Observer

Program Authorized to Offer Degree Computer Science

Date _____

ROCHESTER INSTITUTE OF

TECHNOLOGY

ABSTRACT

GPSense: an algorithmic framework for intelligent sensing at node level in WSN

by Soujanya Soni

Chairperson of the Supervisory Committee: Dr. Rajendra k. Raj

Department of Computer Science

ABSTRACT

We proposed a new Genetic Programming algorithm termed GPSense for use in wireless sensor networks (WSN). Existing algorithms for pattern recognition and data mining in WSNs work offline, i.e. they query the WSN nodes, bring data to the base-station and the mining is done at the base-station. This increases the latency of decision making, enhances decision communication costs and generally leads to non-local decisions. It is generally believed that this paradigm is more power-efficient since sensor nodes are significantly constrained in terms of computing power (processor speed, low-power batteries, limited-memory). We believe that a distributed data mining approach can be evolved where small-footprint mining algorithms can be developed to work on the sensor-nodes thereby improving the current state-of-the-art. GPSense is the first of such in-network data mining frameworks and has the following desirable characteristics:

1. It is designed to work as a distributed algorithm that co-ordinates and exchanges genetic material with collaborating nodes.

2. It is aware of the resource constraints at node level with its footprint being consistently smaller than that of the sensor node's processing capabilities.
3. Its localized nature - enables the entire WSN to make decisions at the node-level instead of aggregating results from long-running continuous queries to the root node for eventual filtering.

In this thesis we describe the GPSense framework and demonstrate its utility based on the results we obtained on a test-bed WSN.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EQUATION

ACKNOWLEDGMENTS

This thesis is the end of my long journey in obtaining my Master degree in Computer Science. I have not traveled alone in this journey. There are some people who made this journey easier with words of encouragement, direction and support.

First a very special thanks to Dr. Ankur Teredesai for the guidance and support he has provided throughout the course of this work. His patience, despite my many, many questions, is greatly appreciated. Thanks to Dr. Rajendra K. Raj for taking important responsibility of chairperson in Dr. Teredesai's absence and for advising me toward completion.

Thanks to Dr. Leon Reznik to read and understand this work. His honest yet considerate criticisms of this work have helped much in improving its quality.

Thanks my family and friends to support me.

Chapter 1

Introduction

Prior works in this area [14-16] have primarily focused on collecting data from sensor nodes, which is then transmitted to a base station for processing. The machine learning algorithms typically reside on the base station. In one way or other, these approaches need to send the data to the base station from each sensor node for the processing. There are following drawbacks of these approaches:

Delay type 1: Time spent in collecting data from all nodes. This delay depends upon following factor:

- WSN size: larger the network more time it takes to collect the data from all nodes.
- Base station configuration.
- Routing protocol in WSN, if any.

Loss of battery Power:

- 75% of battery life goes into radio communication: more use of radio communication, more loss of battery power, hence less node life. In this approach, sensors are sending tons of data to base station, which consumes huge amount of battery life.

Noisy data: Generally, data reached at base station is very noisy because:

- Low battery power of sensor node can results in weak signal strength of a node and because of it data can not reach to base station and add noise in data repository of a base station.
- Broken sensor units.

Scalability of WSN: This approach generally does not support very large WSN because of low bandwidth of these WSN units.

In this work, we are developing a new evolutionary computing paradigm that works within a sensor network itself. We demonstrate the paradigm by implementing Genetic Programming for Sensor Networks and this GP system can be instructed for intelligent-sensing, resource optimized communication strategies, intelligent-routing protocol design, novelty detection, Data mining, etc.

This work is challenging because of various constraints on sensor motes like few kilobytes of RAM, limited battery life, very limited processing power, memory and low bandwidth, etc. For handling these constraints many enhancements need to be done, to reduce the space requirements of the evolutionary algorithm software binary, the data representation and run-time memory requirements etc to name a few.

We believe that our proposed approach has numerous advantages including reduction in consumption of battery power, detecting noisy sensor readings, providing localized decision making to a sensor node, taking a quick reaction for an action/event occurred in the process under inspection, etc to name a few.

For example, A WSN has been deployed to monitor a forest of wood product company to predict tree growth. This WSN run our proposed data mining algorithm on its entire sensor nodes. This WSN learn the prediction model under normal environmental conditions, so as to learn normal behaviors of various sensory attributes at each sensor node itself.
Now, to predict tree growth and yield, expert can query this WSN to predict various attributes depending upon the models of the tree growth. These predictions can be summarized to gain the knowledge about the tree growth. This Knowledge can help a company to make resource allocation decisions to maximize profits.
Similarly, same deployment of WSN in forest can be used to predict the forest fire and its path in the forest to alarm the fire department well before it's too late to take suitable actions.

Chapter 2

Related Work

2.1 Introduction to Wireless sensor Network

2.1.1 An Introduction to Wireless Sensor hardware

The following summary of current MOTE hardware is based on an extract from a dissertation written by Geoff Martin, a Research Associate on the ASTRA project [7]

Over the last few years many different versions of motes have been designed and built by various companies and institutions. The size of these motes varies from roughly the size of a matchbox to the size of a pen tip. MEMS (Micro Electromechanical Systems) technology has been used to miniaturize components with an aim of implementing a mote on a single chip that fits into a volume of no more than a cubic millimeter. These motes have been nicknamed "Smart Dust". In this section various mote designs are looked at which use a variety of communication methods. For this project the Mica2 motes are used.

Mica2 Mote Hardware

The Mica mote was a second generation commercial mote module that is manufactured by Crossbow in the United States. It was mainly used for research and development of low power wireless sensor networks. The Mica mote platform was built around the Atmel. Atmega 128L processor which was capable of running at 4 MHz. The Mica mote had 128Kbytes of flash memory, a 4 Mbit serial flash, 4Kbytes of SRAM and a 4Kbyte EEPROM. TinyOS was used to control the mote and its sensors. The mote was able to communicate with the sensor network via a radio link which operated on the 916 or 433 MHz bands and could carry data at 40 Kbps over distances of up to 100 feet. Power was provided from 2 AA batteries and the device had a battery life of roughly 1 year depending on the application. Sensor

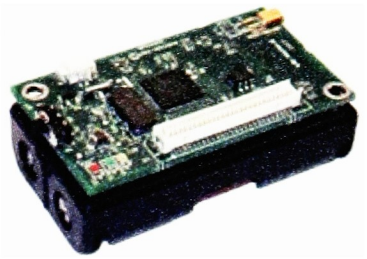

Figure 1: A Mica2 mote [Photo courtesy Crossbow Technology, Inc.]

boards could be attached via a surface mount 51 pin connector that supported analogue input, I2C, SPI, UART and a multiplexed address/data bus.

2.1.2 An Introduction to Wireless Sensor's Programming Environment

TinyOS

TinyOS [8] is an operating system developed at UC Berkeley and was specifically designed for use in sensor networks. It is designed by keeping the needs of sensor networks in multiple ways. First of all, it is designed with power use in mind. Essentially, it is suppose to be a barebones operating system that simply provides the user with the minimum necessary to accomplish the task of programming the mote while at the same time being efficient in terms of both power and processor usage.

One of the most important aspects of TinyOS is its modular design. What this means is that a running TinyOS program is simply a combination of different modules "wired" (this is discussed later) together to comprise the end program.

Another key aspect of TinyOS is its event driven architecture. This aspect was also specifically designed for the type of applications that are running on sensor networks. These applications typically either take periodic measurements or only take measurements in response to either a query or a stimulus in the environment.

TinyOS does not provide a kernel [8]. This means that the programmer deal with the hardware directly. Next, there is no process management. Thus, there is really only one active process at any given time. There is also only a single stack and no virtual memory. Hence, the programmer must be particularly prudent in allocating memory. This is especially true because there is no dynamic memory allocation. Accordingly, all memory allocation is done at compile time.

TinyOS can be downloaded from [8]. This site is also particularly useful for getting specific installation help in terms of a step by step guide. TinyOS can be installed on Windows as well as Linux. Windows installation is accomplished require Cygwin. Also note that it is important to have the proper version of the Java Development Kit installed as well as the associated Java Communications package. These are necessary for TinyOS java programs as well as communications with your motes over the serial interface.

In both installations [8], TinyOS default to install in the directory /opt/tinyos-1.x/. In this directory, there are several directories worth discussing. The /apps directory contains many examples of completed TinyOS modules such as Blink and Surge which are all good starting examples to look at. The /tools/java directory contains java applications used for communications and visualization of your sensor network (including TinyViz). Also check /tools/scripts for the toscheck script to ensure your installation was done properly.

NesC

NesC is essentially the C programming language except with some new key words, a different file structure and naming system. It is developed to support event driven architecture of the TinyOS.

All TinyOS application consists of 2 types of the files:

1) Configuration files(e.g. 'file.nc')

The configuration file starts off with the keyword configuration, thus telling the compiler that we're dealing with a configuration file. This is immediately followed by the name of the module. This file specifies how

and to which other components this particular component is wired to. It also lists any of the interfaces that this module may provide

2) Module file (e.g. 'fileM.nc')

The module file is where the actual implementation takes place. this is where we have to implement the required commands for that interface wiring. Also, if our module is wired to any modules exporting events, this is where we must code our event handlers.

2.2 Genetic Programming

Koza [2] demonstrated a surprising and counter-intuitive result, namely that computers can be programmed by means of natural selection. Specifically, genetic programming is capable of evolving a computer program for solving or approximately solving, a surprising variety of problems from a wide variety of fields. To accomplish this, genetic programming starts with a pool of randomly generated computer programs composed of available programmatic ingredients and genetically breeds the population using the Darwinian principle of survival of the fittest and an analog of naturally occurring genetic crossover (sexual recombination) operation. In other words, genetic programming provides a way to search the space of possible computer programs to find a program that solves, or approximately solves, a problem.

Genetic programming is a domain independent method that genetically breeds populations of computer programs to solve problems by executing the following steps:

1. Generate an initial population of random computer programs composed of the primitive functions and terminals of the problem.
2. Iteratively perform the following sub-steps until the termination criterion has been satisfied:
   2.1. Execute each program in the population and assign it a fitness value according to how well it solves the problem.

2.2. Create a new population of programs by applying the following three primary operations. The operations are applied to program(s) in the population selected with a probability based on fitness (i.e., the fitter the program, the more likely it is to be selected).

- Reproduction: Copy an existing program to the new population.
- Crossover: Create two new off-spring programs for the new population by genetically recombining randomly chosen parts of two existing programs. The genetic crossover (sexual recombination) operation (described below) operates on two parental computer programs and produces two off-spring programs using parts of each parent.
- Mutation: randomly alteration in existing programs, and produces one off-spring programs.
- The single best computer program in the population produced during the run is designated as the result of the run of genetic programming. This result may be solution (or approximate solution) to the problem.

2.2.1 Terminal Set, Function Set and Initial Representation

The terminal and function sets are important components of genetic programming. The terminal and function sets are the alphabet of the programs to be made. The terminal set consists of the variables and constants of the programs.

The functions are several mathematical functions, such as addition, subtraction, division, multiplication and other more complex functions.

The closure property of the function set and terminal set requires that each of the functions in the function set be able to accept, as its arguments, any value and data type that may possibly be returned by any function in the function set and any value and data type that may possibly be assumed by any terminal in the terminal set. That

is, each function in the function set should be well defined and closed for any combination of arguments that it may encounter. The sufficiency property requires that the set of terminals and the set of primitive functions be capable of expressing a solution to the problem.

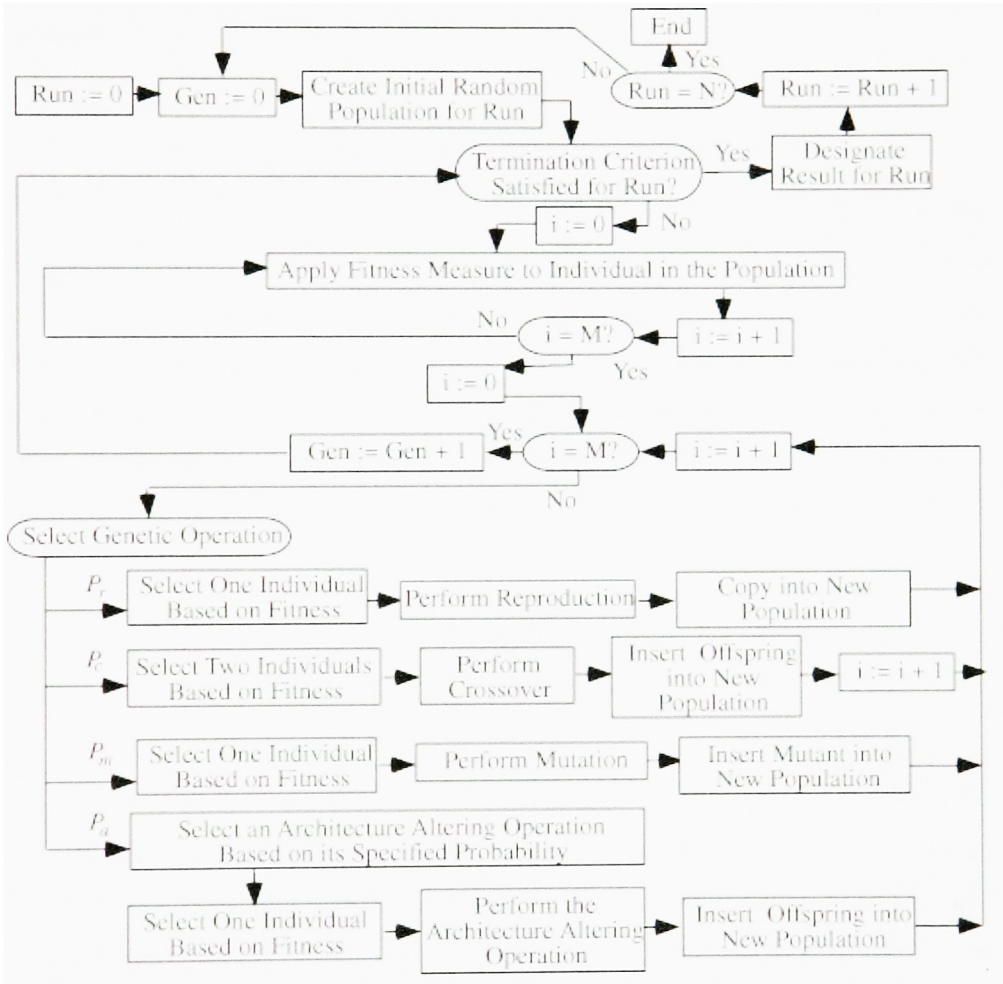

Figure 2: Flowchart of Genetic Programming, Koza [2]

2.2.2 Genetic Operators

Choosing genetic operators is a controversial topic [12]. There are 3 schools of thoughts:

i. only Crossover

ii. only Mutation

iii. Mix of both Crossover and Mutation.

In this work, we are implementing all three possibilities and presenting the comparative results.

i. The Reproduction and Crossover Operations

The two primary genetic operations for modifying the structures undergoing adaptations are Darwinian fitness proportionate. The operation of fitness proportionate reproduction for the genetic programming paradigm is the basic engine of Darwinian reproduction and survival of the fittest. It is a sexual operation in that it operates on only one parental program. The result of this operation is one off-spring program. In this operation, if $f(i,t)$ is the fitness of an individual i in the population M at generation t, the individual i is copied into the next generation with probability

$$\frac{f(i,t)}{\sum_{j=1}^{M} f(j,t)}$$

Equation 1

Note that the operation of fitness proportionate reproduction does not create anything new in the population. It increases or decreases the number of occurrences of individuals already in the population. It improves the average fitness of the population (at the expense of the genetic diversity of the population). To the extent that it increases the number of occurrences of more fit individuals and decreases the number of occurrences of less fit individuals.

The crossover (recombination) operation for the genetic programming paradigm is a sexual operation that starts with two parental programs. Both parents are selected from the population with a probability equal to its normalized fitness.

The result of the crossover operation is two off-spring programs. Unlike fitness proportionate reproduction, the crossover operation creates new individuals in the populations.

The operation begins by randomly and independently selecting one point in each parent using a specified probability distribution (discussed below). Note that the number of points in two parents typically is not equal to each other. As it is seen, the crossover operation is well-defined for any two programs. That is, for any two programs and any two crossover points, the resulting off-springs are always valid computer programs. Offspring contains some traits from each parent.

The crossover fragment for a particular parent is the rooted sub-tree whose root is the crossover point for that parent and where the sub-tree consists of the entire sub-tree lying below the crossover point (i.e., more distant from the root of the original tree). The first off-spring is produced by deleting the crossover fragment of the first parent from the first parent and then impregnating the crossover fragment of the second parent at the crossover point of the first parent. In producing this first off-spring, the first parent acts as the base parent (the female parent) and the second parent acts as the impregnating parent (the male parent). The second off-spring is produced in a symmetric manner. Since entire sub-trees are swapped, this genetic crossover (recombination) operation produces syntactically and semantically valid computer programs as off-spring regardless of which point is selected in either parent.

These two computer programs can be depicted graphically as rooted, point-labeled trees with ordered branches. We are implementing standard sub-tree crossover [2]. The selection of crossover point is performed at random with uniform probability to keep less processing load on sensor processing unit.

It is designed such that new off spring always be in the maximum allowed limit of each individual.

For example, consider the parental computer program $a \times x + b$ and $x2 + a \times b$.

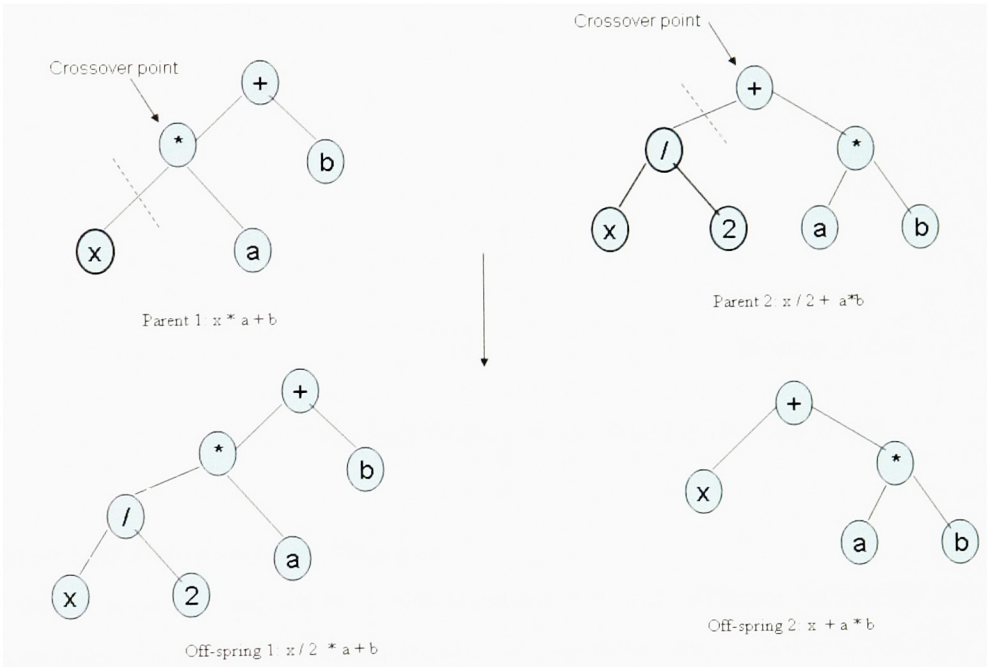

Figure 3: Crossover Operation for Algebraic Equation Manipulation [6]

ii. Mutation

Mutation is another important feature of genetic programming. Two types of mutations are possible. In the first kind, a function can only replace a function or a terminal can only replace a terminal. In the second kind, an entire sub-tree can replace another sub-tree. We are implementing point-mutation in which a function can only replace a function or a terminal can only replace a terminal. We set the mutation probability per node.

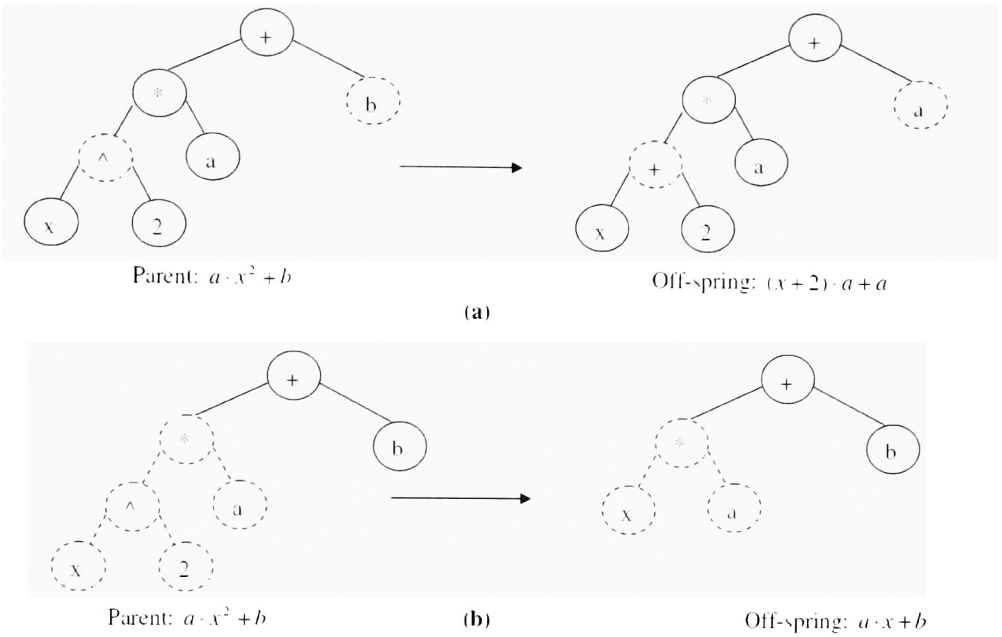

Figure 4: Example of Mutation Operation, (a) Type I & (b) Type II [6]

Mix of both Cross over and Mutation:

In this strategy we use above genetic operator but with different probability of occurrence. For example, if the probability of occurrence for crossover is 0.9 then probability of occurrence for mutation will be 0.1.

Note: Crossover uses more resources then mutation because crossover need to choose two points in two different trees which takes more processing and battery power. Also, crossover needs some memory to store the truncated part of the tree.

iii. The Fitness Measure

The most difficult and most important concept of genetic programming is the fitness function. The fitness function determines how well a program is able to solve the problem. Each individual in a population is assigned a fitness value as a result of its

interaction with the environment. Fitness is the driving force of Darwinian natural selection and genetic algorithms.

Fitness cases provide a basis for evaluating a particular program. The raw fitness of any computer program is the sum, over the fitness cases, of the squares of the distances (taken over all the fitness cases) between the points in the solution space (which is real-valued) returned by the individual program for a given set of arguments and the correct point in the solution space. In particular, the raw fitness $r(i,t)$ of an individual computer program i in the population of size M at any generation t is

$$r(i,t) = \sum_{j=1}^{Ne} [S(i,j) - C(j)]^2 \qquad \text{.... Equation 2}$$

where $S(i,j)$ is the value returned by program i for fitness case j (of Ne fitness cases) and $C(j)$ is the correct value for fitness case j. The closer this sum of distances is to zero, the better the program.

Each raw fitness value is then adjusted (scaled) to produce an adjusted fitness measure $a(i,t)$. The adjusted fitness value is

$$a(i,t) = \frac{1}{(1 + r(i,t))} \qquad \text{... Equation 3}$$

where $r(i,t)$ is the raw fitness for individual i at generation t. Unlike raw fitness, the adjusted fitness is larger for better individuals in the population. Moreover, the adjusted fitness lies between 0 and 1.

Each such adjusted fitness value $a(i,t)$ is then normalized. The normalized fitness value $n(i,t)$ is

$$n(i,t) = \frac{a(i,t)}{\sum_{j=1}^{M} a(j,t)} \qquad \text{... Equation 4}$$

The normalized fitness not only ranges between 0 and 1 and is larger for better individuals in the population, but the sum of the normalized fitness values is 1. Thus, normalized fitness is a probability value.

In a genetic search, each member of a population needs to be evaluated and assigned a fitness value. Obviously, the minimization problem is applied in the whole thesis.

iv. Selection Strategy

Selection Strategy decides "how to choose the individuals in the population that create offspring for the next generation, and how many offspring each create. The purpose of selection is, of course, to emphasize the fitter individuals in the population in hopes that their offspring in turn have even higher fitness. Selection has to be balanced with variation from crossover and mutation (the "exploitation/exploration balance"): too-strong selection means that suboptimal highly fit individuals will take over the population, reducing the diversity needed for further change and progress; too-weak selection will result in too-slow evolution" [11].

As per Adam [4], there are many different techniques which a genetic programming can use to select the individuals to be copied over into the next generation, but listed below are some of the most common methods. Some of these methods are mutually exclusive, but others can be and often are used in combination:

Elitist selection: The most fit members of each generation are guaranteed to be selected.

Fitness-proportionate selection: More fit individuals are more likely, but not certain, to be selected.

Roulette-wheel selection: A form of fitness-proportionate selection in which the chance of an individual's being selected is proportional to the amount by which its fitness is greater or less than its competitors' fitness. (Conceptually, this can be represented as a game of roulette each individual gets a slice of the wheel, but more fit ones get larger slices than less fit ones. The wheel is then spun, and whichever individual "owns" the section on which it lands each time is chosen.)

Scaling selection: As the average fitness of the population increases, the strength of the selective pressure also increases and the fitness function becomes more discriminating. This method can be helpful in making the best selection later on when all individuals have relatively high fitness and only small differences in fitness distinguish one from another.

Rank selection: Each individual in the population is assigned a numerical rank based on fitness, and selection is based on these ranking rather than absolute differences in fitness. The advantage of this method is that it can prevent very fit individuals from gaining dominance early at the expense of less fit ones, which would reduce the population's genetic diversity and might hinder attempts to find an acceptable solution.

Generational selection: The offspring of the individuals selected from each generation become the entire next generation. No individuals are retained between generations.

Steady-state selection: The offspring of the individuals selected from each generation go back into the pre-existing gene pool, replacing some of the less fit members of the previous generation. Some individuals are retained between generations.

Tournament selection: Subgroups of individuals are chosen from the larger population, and members of each subgroup compete against each other. Only one individual from each subgroup is chosen to reproduce. Two individuals are chosen at random from the population. A random number r is then chosen between 0 and 1. If $r < k$ (where k is a parameter, for example 0.75), the fitter of the two individuals is selected to be a parent; otherwise the less fit individual is selected. The two are then returned to the original population and can be selected again.

In this work, we are using Tournament selection because the fitness-proportionate methods described above require two passes through the population at each generation:

1. "one pass to compute the mean fitness (and, for sigma scaling, the standard deviation) and

2. Another pass to compute the expected value of each individual." [11]

This is potentially time consuming procedure and uses more battery power as well. Rank scaling requires sorting the entire population by rank—a potentially time-consuming procedure.

Clearly, other methods take almost twice of processing power then tournament which increases the load on already constrained processing power of sensors and can also result in more consumption of the battery power. Tournament selection is similar to rank selection in terms of selection pressure, but it is computationally more efficient. Tournament group size is 2 because of memory constraint on sensor nodes.

2.3 Symbolic Regression

Symbolic regression (SR) is a technique to find out the relationship among the attributes using the collected data set. This relationship is generally expressed in terms of a mathematical expression e.g. "y = ax + c", etc.

Solving SR using EA come from John Koza who used GP to do same. This class of algorithms is based on Darwinian Theory of evolution and one of its main attributes is that there is no calculated only one solution, but a class of possible solutions at once. This class of possible and acceptable solutions is called "population" Members of this populations are called "individuals" and mathematically said, they represent possible solution

For example, predicting the value of light in lumens based on the current temperature and humidity at the node. Since, a sensor node spends greater amount of energy while taking a light reading compared to taking a temperature reading due to the nature of sensing technology. Hence, if we can accurately correlate light, humidity and temperature at the node levels using a GP model, we can help conserve energy thereby increasing the lifespan of the sensor node. So, the technique of finding out the correlation among the attribute is symbolic regression.

TABLE I
SAMPLE DATA SET AT A SENSOR NODE

Temperature	Humidity	Light
19.1536	45.04	2.03397
19.9884	45.08	2.69964
19.3024	45.08	2.68742

GPSense do symbolic regression on this data set and try to come up with an expression in term of Temperature and humidity. Such that,

$$\text{Light} = f(\text{temperature}, \text{humidity})$$

Using above expression, it possible to predict the value of light, if we know temperature and humidity.

2.4 Previous work

Currently there are very few in-network implementations of machine learning algorithms hence it is difficult to cite work that is truly related to our efforts. In a related effort Johnson et al [1] developed a distributed algorithm they termed Broadcast Distributed Parallel GA (BDP-GA) that has a low footprint. Encouraged by the results they obtained in a simulated environment; it is our goal to develop the GPSense algorithm that performs in-network GP on sensor nodes.

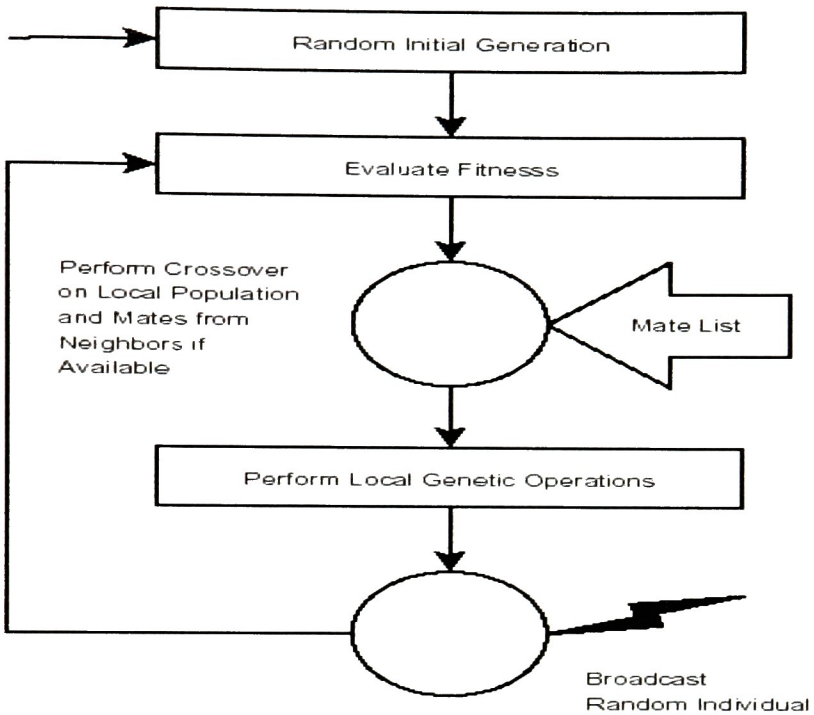

Figure 5: GA running on a sensor node [1]

Above figure clearly explain the BDP-GA algorithm developed by Johnson et al [1] on simulator.

There are 2 extensions to the approach by the work Johnson et al [1] which we attempt in this thesis:

1. Implement similar distributed in-network algorithm on a real sensor node within a WSN instead of a simulator.
2. Use this new algorithm for prediction of a sensory attribute.

Chapter 3

Functional Specification of GPSense

GPSense is a Genetic programming based in-network distributed EC algorithm for a WSN. It runs on each sensor node and shares its local genetic information with other neighboring sensor nodes over wireless. GPSense does symbolic regression at each sensor node on sensed data. Symbolic regression is a technique to discover a function which can represent the relationship fits best in finite set of data because it is well known fact under signal processing field that various sensory attributes obey some relationship depending upon the process under inspection.

For example, predicting the value of light in lumens based on the current temperature and humidity at the node. Since, a sensor node spends greater amount of energy while taking a light reading compared to taking a temperature reading due to the nature of sensing technology. Hence, if we can accurately correlate light, humidity and temperature at the node levels using a GP model, we can help conserve energy thereby increasing the lifespan of the sensor node.

TABLE IIIII
SAMPLE DATA SET AT A SENSOR NODE

Temperature	Humidity	Light
19.1536	45.04	2.03397
19.9884	45.08	2.69964
19.3024	45.08	2.68742

GPSense use symbolic regression on this data set and try to come up with an expression in term of Temperature and humidity. Such that,

$$\text{Light} = f(\text{temperature}, \text{humidity})$$

This genetic information from each generation of GP at a sensor node is shared with other neighboring nodes asynchronously over wireless. This sharing of local genetic information with other neighboring node greatly improves the convergence of the GP; otherwise GP running alone on a sensor node would likely take too long to be of any use.

In the same way, every node receives the genetic material from other neighboring nodes. After receiving the genetic material local GP does local reproduction with local and remote genetic material and then calculate the fitness measure for the entire population.

This process is repeated till GPSense gets converge or limit on number of evolutions get over for GP. This evolution limit can set manually in the GPSense program.

Next, suppose we want to predict the value of light at the node, given the values for temperature and humidity being known. We can use above expression to predict the value of light.

3.1 Implementation details

1) Terminal Set and Function Set

The populations can contain three types of chromosomes: algebraic, space-separated, and binary strings. Algebraic chromosomes are algebraic expressions, which is applied in this problem. The genetic programming module contains a set of primitive functions like +, -, *, / a terminal set consisting of floating point temperature, Light etc. It performs symbolic regression on the data collected by the each sensor node to extract functional representation of the data because it is well known fact under signal processing field that various sensory attributes obey some relationship depending upon the process under inspection [10].

2) Parameters for Controlling Runs

The GPSense paradigm is controlled by two major numerical parameters, i.e., the population size and the maximum number of generations. Other minor numerical parameters include the probability of crossover and mutation. The maximum population size allowed on GPSense is 150.

3) Genetic Operators

Choosing genetic operators is a controversial topic [12]. There are 3 schools of thoughts:

iv. only Crossover

v. only Mutation

vi. Mix of both Crossover and Mutation.

In this work, we are implementing all three possibilities as explained below:

Cross over:

We are implementing standard sub-tree crossover [2]. The selection of crossover point is performed at random with uniform probability to keep less processing load on sensor processing unit.

It is designed such that new off spring always be in the maximum allowed limit of each individual.

Mutation:

We are implementing point-mutation in which a function can only replace a function or a terminal can only replace a terminal. We set the mutation probability per node.

Mix of both Cross over and Mutation:

In this strategy we use above genetic operator but with different probability of occurrence. For example, if the probability of occurrence for crossover is 0.9 then probability of occurrence for mutation will be 0.1.

Note: Crossover uses more resources then mutation because crossover need to choose two points in two different trees which takes more processing and battery power. Also, crossover needs some memory to store the truncated part of the tree.

4) Selection Strategy

In this work, we are using Tournament selection because the fitness-proportionate methods described above require two passes through the population at each generation:

3. "one pass to compute the mean fitness (and, for sigma scaling, the standard deviation) and

4. Another pass to compute the expected value of each individual." [11]

This is potentially time consuming procedure and uses more battery power as well. Rank scaling requires sorting the entire population by rank—a potentially time-consuming procedure.

Clearly, other methods take almost twice of processing power then tournament which increases the load on already constrained processing power of sensors and can also result in more consumption of the battery power. Tournament selection is similar to rank selection in terms of selection pressure, but it is computationally more efficient. Tournament group size is 2 because of memory constraint on sensor nodes.

Format of message, we are using to share local genetic material within the network look like:

```
typedef struct MyIntMsgType {
char *data;            // Expression showing relationship between attributes.
uint16_t src;          // Address of the sender node
} MyIntMsgType;
```

This genetic information is broadcasted by the each node at each generation and same information is received by each node available in the network and added to local population of the node for breeding.

Before starting the training period data on various attributes (like light, temperature etc) is collected by each sensor node from its environment and will be stored at each sensor node in the form of a table. We are planning to cluster the collected value and store the average of 20 values for each attribute instead of storing all 20 values. This help in covering more variant data space with low in-memory data.

GPSense algorithm

At each Node X:

```
1. Initialization
2. Start Timer
3. Collect training data on light, temperature and
   voltage from environment
4. Pop[POPSIZE]  create_random_population()
5. Measure fitness of each individual in this
   population
6. bestexp  individual with best fitness value.
7. if Timer fired
8. then Broadcast bestexp
9. if fitness (bestexp) > THRESHOLD_FITNESS
10. then
11.   Stop training the model.
12.   Store bestexp in EEPROM as a result
13. else
14.    Post get_new_population ()
15. end if
16. if Message received
```

```
17.   Unpack the message and took out genetic
   material in the message and add it to the local
   population
18. end if
19. end if
20. end

1. Task get_new_population()
2. if gen < GENERATION
3. then RUN:=0
4. for I □ 0  to N   DO
5. begin
6. Gen □ 1
7. while not (Terminate condition for run)
8. begin
9. Measure fitness of each individual in this
   population
10. bestexp □ individual with best fitness value.
11. for J □ 1 to POPSIZE    DO
12. begin
13. OP □ Select Genetic Operator
14. Select two individuals based on fitness using
15. tournament Selection and Perform operation 'OP'
16. Insert new two offspring into new population
17. J □ J+1
18. end
19. end
20. Measure fitness of each individual in this
   population
21. bestexp □ individual with best fitness value.
22. Gen □ Gen+1
23. end
24. Designate result for Run
25. RUN □ RUN+1
```

```
26. else
27. Store bestexp in the EEPROM
28. Stop training the model
29. end if
30. end
```

Note: A task is just like any other function except it runs in background. The post operation places the task on an internal task queue which is processed in FIFO order. When a task is executed, it runs to completion before the next task is run.

3.2 Complexity of GPSense

GPSense works in three phases as follows:

1. Collect data from environment
2. GP Training process
3. Coordination

Collect data from environment:

Collecting data from environment depends upon the timer frequency as well as the amount of data we want to collect.

Suppose it collect 'n' sample of data from the environment at the frequency of 1 sample/second, then it takes n* 1 = n seconds to collects n-samples.

Therefore, to collect n-sample at frequency 'f',

$$\text{Data collection takes time} = c_1 * (n * f) \text{ seconds}$$

Where c_1 = cost to collect one set of data sample from environment.

GP Training process:

Training GP involves following steps:

1. Creating new population
2. Calculating fitness of each individual in the population.

Above steps are repeated for each generation.

Suppose, 'P' is the population size and 'G' is the number of generation then,

Time taken to train a GP = $(c_2 * P + c_3 * P) * G$

Where,

c_2 = cost to create an individual

c_3 =cost to create to calculate fitness of an individual

Coordination

At each generation, GPSense broadcast best individual. Time taken to broadcast one individual depend upon the size of an individual. Since maximum size for individual is fixed in GPSense, we can estimate the worst case time requirement to broadcast an individual. If time taken to broadcast an individual is 't' sec. Then, total time taken to broadcast all individual from all generation (G) = (G * t) seconds

Therefore,

Total processing time for GPSense = $c_1*(n * f) + (c_2 * P + c_3 * P) * G + (G * t)$

Clearly, Data collection and Coordination are of liner order but Training GP is a Quadratic function which is the bottle neck of whole GPSense algorithm and hence, GPSense is also quadratic in complexity.

3.3 Space requirement of the GPSense

3.3.1 On a single mote

Since TinyOS doesn't support Dynamic memory allocation, size of program binaries depicts total memory requirement of GPSense at run time on single node. Therefore,

Memory requirement for GPSense = Program Binary size, β

Equation 5

Where,

$$\beta = \gamma + \delta + \tau + C$$

Equation 6

□ = Memory required to store population at each generation

= Size of each individual * population size, M

□ = Mate list coming from neighboring nodes. This varies for each sensor node depending upon its location in a WSN i.e. if it is has many neighboring node. It will get big mate list and an isolated node will not has any mate list.

= Amount of memory needed to store the data for training the model.

C = others, like temp variable, program statements, etc

3.3.2 With in a Wireless sensor network

For the WSN with 'n' number of motes, total memory requirement with in this network can be calculated using following equations:

$$\text{GPSense memory requirement with in the network} = \beta \times n$$

Equation 7

Where

= Memory requirement of GPSense on a single node.

Chapter 4

Experiments and Results

All the tests were executed on Mica2 motes. The specification for Mica2 motes is as follow:

- Constrained Processor : 4Mhz, 8 bit operation
- Little memory: 128Kb Flash, 4Kb RAM and 4Kb EEPROM
- Low bit rate communication: 40Kbits/seconds
- Short range transmission: ~100 feet
- Low Energy: Runs on 2 AA batteries.

4.1 Space requirement of the GPSense

4.1.1 On a single mote

As per the equation 3,

Memory requirement for GPSense at a Mote = 3978bytes (RAM) and 17834bytes(ROM)

□ = Program Binaries ≈ **3978bytes (RAM)**

This includes:

□ = Memory required to store population at each generation

= Size of each individual × population size, P

= $(2^{depth_{max}} - 1) \times$ sizeof (tree node) × P

= $(2^4 - 1) \times 1 \times 170 =$ **2550 bytes**

□ = Mate list coming from neighboring nodes. This varies for each sensor node. Depending upon its location in a WSN i.e. if it is has many neighboring node. It will get big mate list and an isolated node will not has any mate list. Therefore,

= MAX (_sublist_i | i = _ (size of the mate list distributed by neighboring nodes))

= $(2^{depth_{max}} - 1) \times$ sizeof (tree node) × max_limit

= $(2^{4} - 1) \times 1 \times 10 =$ **150 bytes**

□ = Amount of memory needed to store the data for training the model. We are collecting data on three attributes each as of type float.

1. Temperature
2. Light
3. Voltage.

We are taking average of 10 collected samples as an entry in a data set for each attributes. And in total we are taking 100 data entries. Therefore,

□ **= 90 × 3 × 4 = 1080 bytes**

C = 3978 bytes – (1080 + 150 + 2550) bytes = 198 bytes.

4.1.2 With in the Wireless sensor network

As per equation 9:

GPSense memory requirement with in the network = $\beta \times n$

= **(3978 × n) bytes**

4.2 Test GP convergence on a mote:

For testing, Mica2 motes with a sensor board are used. This senor board is capable of sensing attributes like light, temperature, humidity, etc from the environment and this data can be stored on memory available on the mote. These attributes are terminals for the GP run. In terms of problem, GP is designed to solve symbolic regression on the sensed data. Each solution tree in the GP run represents the relationship between sensed attributes.

In this test, a mica2 mote was used which running GPSense at every generation of GP run, mote broadcast the best individual with its fitness value which is collected at the base station for testing. If the last received individual from a node shows error < 10% or last individual is coming from the last generation of the GP, and then GP is consider converged. The parameter and value used in GP are listed in table _.

Parameters	Value
Problem: Data set:	Symbolic regression Sensed data (temperature, light, humidity)
Terminal set: Function set:	x1, x2, x3 +,-,*,/
Population size: Experimental Choices For: Crossover probability: Mutation Probability: Selection: Termination criterion: Maximum Generation: Maximum depth: Initialization method:	150 Varies (0 to 1) 1 – crossover Tournament Error < 10% Varies(10 to 60) 4 Grow

Table 4 PARAMETERS AND VALUE: TEST CONVERGENCE OF GP ON A SENSOR NODE

Since choosing genetic operators is a controversial topic [8], we ran our tests with different probability of occurrences for genetic operators e.g. if in any test probability of occurrence for crossover is 0.4 then probability of occurrence for mutation will be 0.6. Also, we are trying to show the effect of number of generation over the maximum fitness of individuals attained with various probabilities of occurrence of genetic operators. Fig.1 shows the result we obtained for these tests.

Generation =10

Test	Mutation rate	Cross over rate	Maximum fitness
T_1	0	100	16.6037
T_2	5	95	16.6037
T_3	10	90	16.6037
T_4	15	85	16.6034
T_5	20	80	16.6037
T_6	25	75	16.6037
T_7	30	70	16.6037
T_8	35	65	16.6037
T_9	40	60	16.6034
T_{10}	45	55	16.6034
T_{11}	50	50	16.6034
T_{12}	55	45	16.6037
T_{13}	60	40	16.5341
T_{14}	65	35	16.6037
T_{15}	70	30	16.6037
T_{16}	75	25	16.6037
T_{17}	80	20	16.5341
T_{18}	85	15	16.3089
T_{19}	90	10	16.6034
T_{20}	95	5	16.5341
T_{21}	100	0	16.5341

Generation =20

Test	Mutation rate	Cross over rate	Maximum fitness
T_1	0	100	15.7995
T_2	5	95	16.3037
T_3	10	90	16.2684
T_4	15	85	15.7995
T_5	20	80	16.3037
T_6	25	75	15.7913
T_7	30	70	16.2371
T_8	35	65	16.2286
T_9	40	60	16.2381
T_{10}	45	55	16.239
T_{11}	50	50	16.2812
T_{12}	55	45	16.2379
T_{13}	60	40	16.2346
T_{14}	65	35	16.2292
T_{15}	70	30	16.2353
T_{16}	75	25	16.2319
T_{17}	80	20	16.2141
T_{18}	85	15	16.2332
T_{19}	90	10	16.2812
T_{20}	95	5	16.2335
T_{21}	100	0	16.2234

Generation =30

Test	Mutation rate	Cross over rate	Maximum fitness
T_1	0	100	16.6037
T_2	5	95	16.6037
T_3	10	90	16.2285
T_4	15	85	16.4422
T_5	20	80	16.2381
T_6	25	75	16.6037
T_7	30	70	16.2379
T_8	35	65	16.2199
T_9	40	60	16.2812
T_{10}	45	55	16.2108
T_{11}	50	50	16.2812
T_{12}	55	45	16.2812
T_{13}	60	40	16.2334
T_{14}	65	35	16.1449
T_{15}	70	30	16.2335
T_{16}	75	25	16.2201
T_{17}	80	20	16.2283
T_{18}	85	15	16.2301
T_{19}	90	10	16.2308
T_{20}	95	5	16.2062
T_{21}	100	0	16.2057

Generation=40

Test	Mutation rate	Cross over rate	Maximum fitness
T_1	0	100	16.6037
T_2	5	95	16.4499
T_3	10	90	16.6037
T_4	15	85	16.4506
T_5	20	80	16.6034
T_6	25	75	16.6037
T_7	30	70	16.2635
T_8	35	65	16.6037
T_9	40	60	16.6031
T_{10}	45	55	16.233
T_{11}	50	50	16.2333
T_{12}	55	45	16.1713
T_{13}	60	40	16.2335
T_{14}	65	35	16.2312
T_{15}	70	30	16.2259
T_{16}	75	25	16.2245
T_{17}	80	20	16.2332
T_{18}	85	15	16.2312
T_{19}	90	10	16.2029
T_{20}	95	5	16.2312
T_{21}	100	0	16.2329

Generation=50

Test	Mutation rate	Cross over rate	Maximum fitness
T_1	0	100	16.6037
T_2	5	95	16.6037
T_3	10	90	16.4393
T_4	15	85	16.6037
T_5	20	80	16.2807
T_6	25	75	16.2354
T_7	30	70	16.2332
T_8	35	65	16.6037
T_9	40	60	16.239
T_{10}	45	55	16.2381
T_{11}	50	50	16.2332
T_{12}	55	45	16.2335
T_{13}	60	40	16.0557
T_{14}	65	35	16.081
T_{15}	70	30	16.1939
T_{16}	75	25	16.2332
T_{17}	80	20	16.2293
T_{18}	85	15	15.7913
T_{19}	90	10	16.2161
T_{20}	95	5	13.4606
T_{21}	100	0	14.4414

Generation =60

Test	Mutation rate	Cross over rate	Maximum fitness
T_1	0	100	15.7995
T_2	5	95	16.2812
T_3	10	90	15.5477
T_4	15	85	14.4417
T_5	20	80	16.2682
T_6	25	75	16.2382
T_7	30	70	16.1896
T_8	35	65	14.4936
T_9	40	60	14.4414
T_{10}	45	55	13.6946
T_{11}	50	50	14.4414
T_{12}	55	45	14.2375
T_{13}	60	40	14.2822
T_{14}	65	35	15.7359
T_{15}	70	30	15.7145
T_{16}	75	25	14.4414
T_{17}	80	20	10.5116
T_{18}	85	15	13.3795
T_{19}	90	10	11.3436
T_{20}	95	5	14.1117
T_{21}	100	0	13.8946

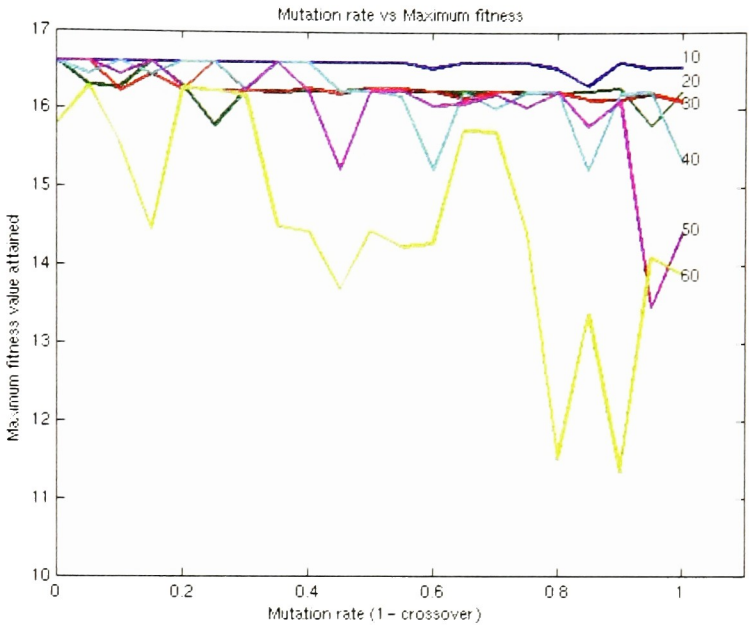

Figure 6. Plot showing maximum fitness attained over probability of mutation where numbers at the end of each curve show the number of generation

From the above results, it can be concluded that having higher mutation rate and more number of generation are helping GP to find out better individuals at node level.

4.3 Test for accuracy:

In this test, two mica2 motes (A and B) were used both were placed side by side so as to let them collect same data from the environment. Node 'B' sent out whole data to the base station for testing purpose while Node 'A' run GPSense and start building the model on collected data at the node level. At every generation of GP run, node 'A' broadcast the best individual with its fitness value which is collected at the base station for testing. If the last received individual from a node shows error < 10% or last individual is coming from the last generation of the GP, and then GP is consider converged. The parameter and value used in GP are listed in table

Parameters	Value
Problem: Data set:	Symbolic regression Sensed data (temperature, light, humidity)
Terminal set:	x1, x2, x3

Function set:	+,-,*,/
Population size:	150
Experimental Choices For:	
Crossover probability:	Varies (0 to 1)
Mutation Probability:	1 – crossover
Selection:	Tournament
Termination criterion:	Error < 10%
Maximum Generation:	Varies(10 to 60)
Maximum depth:	4
Initialization method:	Grow

Table 5:PARAMETERS AND VALUE : TEST FOR ACCURACY

We ran tests with different probability of occurrences for genetic operators e.g. if in any test probability of occurrence for crossover is 0.4 then probability of occurrence for mutation will be 0.6

At the base station, we used the received individual to predict the value for temperature using the test data collected from node 'B' Then, we calculated the squared mean error for such test.

Also, we are trying to show the effect of number of generation over the mean squared error between actual and predicted temperature value with various probabilities of occurrence of genetic operators. Fig. 7 shows the result we obtained for these tests.

i. Predicting Temperature given Voltage and Light at a sensor node

Generation=10

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0721
T_2	5	95	0.0721
T_3	10	90	0.0721
T_4	15	85	0.0721
T_5	20	80	0.0721
T_6	25	75	0.0499
T_7	30	70	0.0499
T_8	35	65	0.0499
T_9	40	60	0.0499
T_{10}	45	55	0.0499
T_{11}	50	50	0.0499
T_{12}	55	45	0.0499
T_{13}	60	40	0.0499
T_{14}	65	35	0.0499
T_{15}	70	30	0.0499
T_{16}	75	25	0.0499
T_{17}	80	20	0.0499
T_{18}	85	15	0.0499
T_{19}	90	10	0.0499
T_{20}	95	5	0.0499
T_{21}	100	0	0.0499

Generation=20

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0721
T_2	5	95	0.0721
T_3	10	90	0.0721
T_4	15	85	0.0721
T_5	20	80	0.0721
T_6	25	75	0.0721
T_7	30	70	0.0499
T_8	35	65	0.0499
T_9	40	60	0.0499
T_{10}	45	55	0.0499
T_{11}	50	50	0.0499
T_{12}	55	45	0.0499
T_{13}	60	40	0.0499
T_{14}	65	35	0.0499
T_{15}	70	30	0.0499
T_{16}	75	25	0.0499
T_{17}	80	20	0.0499
T_{18}	85	15	0.0499
T_{19}	90	10	0.0499
T_{20}	95	5	0.0499
T_{21}	100	0	0.0499

Generation=30

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0721
T_2	5	95	0.0721
T_3	10	90	0.0721
T_4	15	85	0.0721
T_5	20	80	0.0499
T_6	25	75	0.0499
T_7	30	70	0.0499
T_8	35	65	0.0499
T_9	40	60	0.0499
T_{10}	45	55	0.0499
T_{11}	50	50	0.0344
T_{12}	55	45	0.0344
T_{13}	60	40	0.0499
T_{14}	65	35	0.0499
T_{15}	70	30	0.0432
T_{16}	75	25	0.0432
T_{17}	80	20	0.0517
T_{18}	85	15	0.0517
T_{19}	90	10	0.0401
T_{20}	95	5	0.0401
T_{21}	100	0	0.0167

Generation=40

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0721
T_2	5	95	0.0721
T_3	10	90	0.0721
T_4	15	85	0.0721
T_5	20	80	0.0432
T_6	25	75	0.0432
T_7	30	70	0.0432
T_8	35	65	0.0432
T_9	40	60	0.0432
T_{10}	45	55	0.0432
T_{11}	50	50	0.0235
T_{12}	55	45	0.0235
T_{13}	60	40	0.0235
T_{14}	65	35	0.0235
T_{15}	70	30	0.0519
T_{16}	75	25	0.0519
T_{17}	80	20	0.0519
T_{18}	85	15	0.0519
T_{19}	90	10	0.0238
T_{20}	95	5	0.0238
T_{21}	100	0	0.0519

Generation=50

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0721
T_2	5	95	0.0721
T_3	10	90	0.061
T_4	15	85	0.061
T_5	20	80	0.0721
T_6	25	75	0.0721
T_7	30	70	0.062
T_8	35	65	0.062
T_9	40	60	0.0344
T_{10}	45	55	0.0344
T_{11}	50	50	0.0364
T_{12}	55	45	0.0344
T_{13}	60	40	0.0344
T_{14}	65	35	0.0344
T_{15}	70	30	0.0344
T_{16}	75	25	0.0344
T_{17}	80	20	0.0225
T_{18}	85	15	0.0225
T_{19}	90	10	0.0344
T_{20}	95	5	0.0344
T_{21}	100	0	0.0464

Generation=60

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0721
T_2	5	95	0.0721
T_3	10	90	0.0491
T_4	15	85	0.0491
T_5	20	80	0.051
T_6	25	75	0.051
T_7	30	70	0.0344
T_8	35	65	0.0344
T_9	40	60	0.0344
T_{10}	45	55	0.0344
T_{11}	50	50	0.0485
T_{12}	55	45	0.0485
T_{13}	60	40	0.0485
T_{14}	65	35	0.0485
T_{15}	70	30	0.0168
T_{16}	75	25	0.0168
T_{17}	80	20	0.0168
T_{18}	85	15	0.0168
T_{19}	90	10	0.0419
T_{20}	95	5	0.0419
T_{21}	100	0	0.0425

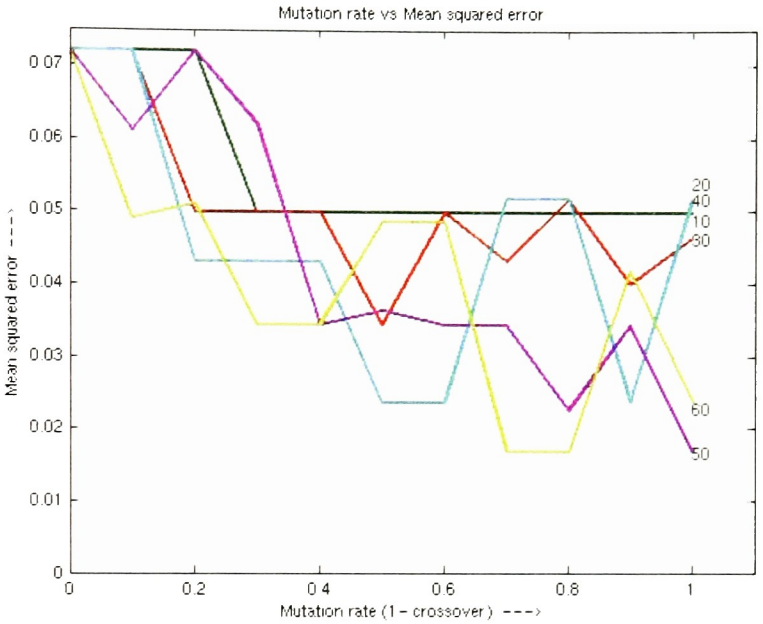

Figure. 7. Plot showing Mean squared error over probability of mutation where numbers at the end of each curve show the number of generation (Predicting Temperature given Voltage and Light)

ii. Predicting Voltage given Temperature and Light at a sensor node

Generation=10

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0209
T_2	5	95	0.0209
T_3	10	90	0.0024
T_4	15	85	0.0085
T_5	20	80	0.0085
T_6	25	75	0.0085
T_7	30	70	0.0033
T_8	35	65	0.0033
T_9	40	60	0.0039
T_{10}	45	55	0.0039
T_{11}	50	50	0.0039
T_{12}	55	45	0.0039
T_{13}	60	40	0.0067
T_{14}	65	35	0.0067
T_{15}	70	30	0.0067
T_{16}	75	25	0.0017
T_{17}	80	20	0.0017
T_{18}	85	15	0.0017
T_{19}	90	10	0.0010
T_{20}	95	5	0.0010
T_{21}	100	0	0.0010

Generation=20

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0209
T_2	5	95	0.0209
T_3	10	90	0.0209
T_4	15	85	0.0081
T_5	20	80	0.0081
T_6	25	75	0.0209
T_7	30	70	0.0056
T_8	35	65	0.0056
T_9	40	60	0.0034
T_{10}	45	55	0.0056
T_{11}	50	50	0.0034
T_{12}	55	45	0.0034
T_{13}	60	40	0.0023
T_{14}	65	35	0.0023
T_{15}	70	30	0.0034
T_{16}	75	25	0.0014
T_{17}	80	20	0.0014
T_{18}	85	15	0.0014
T_{19}	90	10	0.0025
T_{20}	95	5	0.0014
T_{21}	100	0	0.0025

Generation=30

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0209
T_2	5	95	0.0047
T_3	10	90	0.0209
T_4	15	85	0.0047
T_5	20	80	0.0047
T_6	25	75	0.0006
T_7	30	70	0.0006
T_8	35	65	0.0004
T_9	40	60	0.0004
T_{10}	45	55	0.0008
T_{11}	50	50	0.0004
T_{12}	55	45	0.0004
T_{13}	60	40	0.0008
T_{14}	65	35	0.0008
T_{15}	70	30	0.0005
T_{16}	75	25	0.0005
T_{17}	80	20	0.0005
T_{18}	85	15	0.0005
T_{19}	90	10	0.0003
T_{20}	95	5	0.0003
T_{21}	100	0	0.0003

Generation=40

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0047
T_2	5	95	0.0047
T_3	10	90	0.0047
T_4	15	85	0.0014
T_5	20	80	0.0014
T_6	25	75	0.0014
T_7	30	70	0.0014
T_8	35	65	0.0006
T_9	40	60	0.0014
T_{10}	45	55	0.0006
T_{11}	50	50	0.0006
T_{12}	55	45	0.0004
T_{13}	60	40	0.0005
T_{14}	65	35	0.0004
T_{15}	70	30	0.0004
T_{16}	75	25	0.0005
T_{17}	80	20	0.00004
T_{18}	85	15	0.00004
T_{19}	90	10	0.00004
T_{20}	95	5	0.00005
T_{21}	100	0	0.00005

Generation=50

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0047
T_2	5	95	0.0047
T_3	10	90	0.0024
T_4	15	85	0.0024
T_5	20	80	0.0024
T_6	25	75	0.0024
T_7	30	70	0.0012
T_8	35	65	0.0012
T_9	40	60	0.0012
T_{10}	45	55	0.0014
T_{11}	50	50	0.0006
T_{12}	55	45	0.0005
T_{13}	60	40	0.0006
T_{14}	65	35	0.0004
T_{15}	70	30	0.0004
T_{16}	75	25	0.00025
T_{17}	80	20	0.00022
T_{18}	85	15	0.00025
T_{19}	90	10	0.00005
T_{20}	95	5	0.00005
T_{21}	100	0	0.00006

Generation=60

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0047
T_2	5	95	0.0047
T_3	10	90	0.0014
T_4	15	85	0.0047
T_5	20	80	0.0014
T_6	25	75	0.0014
T_7	30	70	0.0006
T_8	35	65	0.0005
T_9	40	60	0.0006
T_{10}	45	55	0.0006
T_{11}	50	50	0.0005
T_{12}	55	45	0.0004
T_{13}	60	40	0.0001
T_{14}	65	35	0.0002
T_{15}	70	30	0.0001
T_{16}	75	25	0.0001
T_{17}	80	20	0.00005
T_{18}	85	15	0.00005
T_{19}	90	10	0.00004
T_{20}	95	5	0.00005
T_{21}	100	0	0.00002

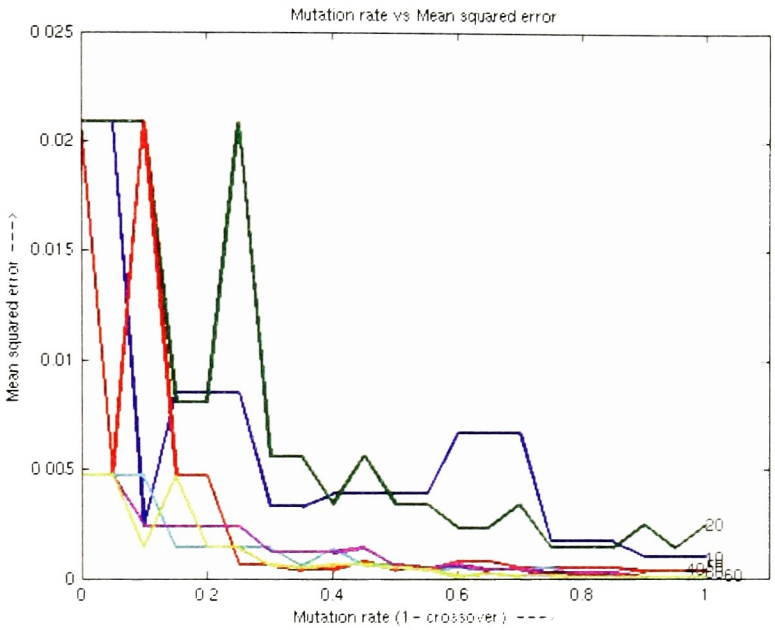

Figure 8. Plot showing Mean squared error over probability of mutation where numbers at the end of each curve show the number of generation (Predicting Voltage given temperature and Light)

iii. Predicting Light given Temperature and voltage at a sensor node

Generation=10

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0317
T_2	5	95	0.0317
T_3	10	90	0.0317
T_4	15	85	0.0317
T_5	20	80	0.0317
T_6	25	75	0.0317
T_7	30	70	0.0317
T_8	35	65	0.0317
T_9	40	60	0.0317
T_{10}	45	55	0.1263
T_{11}	50	50	0.1263
T_{12}	55	45	0.1263
T_{13}	60	40	0.1263
T_{14}	65	35	0.1263
T_{15}	70	30	0.1263
T_{16}	75	25	0.1263
T_{17}	80	20	0.1928
T_{18}	85	15	0.1928
T_{19}	90	10	0.1263
T_{20}	95	5	0.1263
T_{21}	100	0	0.1263

Generation=20

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0317
T_2	5	95	0.0317
T_3	10	90	0.1263
T_4	15	85	0.0317
T_5	20	80	0.1263
T_6	25	75	0.0317
T_7	30	70	0.1263
T_8	35	65	0.0317
T_9	40	60	0.1263
T_{10}	45	55	0.0317
T_{11}	50	50	0.1263
T_{12}	55	45	0.1263
T_{13}	60	40	0.1263
T_{14}	65	35	0.1263
T_{15}	70	30	0.1919
T_{16}	75	25	0.1919
T_{17}	80	20	0.1752
T_{18}	85	15	0.1919
T_{19}	90	10	0.1919
T_{20}	95	5	0.1919
T_{21}	100	0	0.1752

Generation=30

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0209
T_2	5	95	0.2033
T_3	10	90	0.0209
T_4	15	85	0.2033
T_5	20	80	0.2033
T_6	25	75	0.0209
T_7	30	70	0.0209
T_8	35	65	0.0209
T_9	40	60	0.2033
T_{10}	45	55	0.0209
T_{11}	50	50	0.1806
T_{12}	55	45	0.1806
T_{13}	60	40	0.2033
T_{14}	65	35	0.1806
T_{15}	70	30	0.8764
T_{16}	75	25	0.8764
T_{17}	80	20	0.8764
T_{18}	85	15	0.1806
T_{19}	90	10	0.1806
T_{20}	95	5	0.1806
T_{21}	100	0	0.1806

Generation=40

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0209
T_2	5	95	0.0209
T_3	10	90	0.0249
T_4	15	85	0.0209
T_5	20	80	0.0239
T_6	25	75	0.0239
T_7	30	70	0.0209
T_8	35	65	0.0229
T_9	40	60	0.0209
T_{10}	45	55	0.0239
T_{11}	50	50	0.0239
T_{12}	55	45	0.0948
T_{13}	60	40	0.0948
T_{14}	65	35	0.0239
T_{15}	70	30	0.0239
T_{16}	75	25	0.0239
T_{17}	80	20	0.0948
T_{18}	85	15	0.0948
T_{19}	90	10	0.0229
T_{20}	95	5	0.0948
T_{21}	100	0	0.0948

Generation=50

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0209
T_2	5	95	0.0209
T_3	10	90	0.0209
T_4	15	85	0.0239
T_5	20	80	0.0239
T_6	25	75	0.0209
T_7	30	70	0.0239
T_8	35	65	0.0239
T_9	40	60	0.0209
T_{10}	45	55	0.0239
T_{11}	50	50	0.1898
T_{12}	55	45	0.6509
T_{13}	60	40	0.1898
T_{14}	65	35	0.6509
T_{15}	70	30	0.1898
T_{16}	75	25	0.6509
T_{17}	80	20	0.1898
T_{18}	85	15	0.6509
T_{19}	90	10	0.6009
T_{20}	95	5	0.6509
T_{21}	100	0	0.6009

Generation=60

Test	Mutation rate	Cross over rate	Mean Square Error
T_1	0	100	0.0209
T_2	5	95	0.6557
T_3	10	90	0.0209
T_4	15	85	0.0209
T_5	20	80	0.6557
T_6	25	75	1.4085
T_7	30	70	0.6557
T_8	35	65	0.6557
T_9	40	60	1.4085
T_{10}	45	55	0.6557
T_{11}	50	50	1.4085
T_{12}	55	45	0.6557
T_{13}	60	40	1.4085
T_{14}	65	35	0.6557
T_{15}	70	30	1.4085
T_{16}	75	25	0.0288
T_{17}	80	20	0.6557
T_{18}	85	15	0.6557
T_{19}	90	10	0.0288
T_{20}	95	5	0.6557
T_{21}	100	0	0.0288

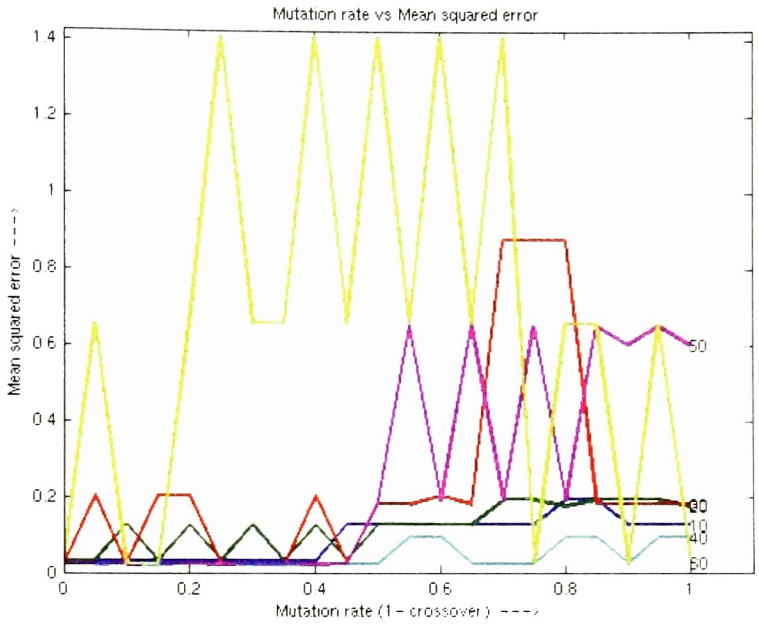

Figure 9.Plot showing Mean squared error over probability of mutation where numbers at the end of each curve show the number of generation (Predicting Light given voltage and temperature)

Chapter 5

Conclusion and Future work

In this work, we developed a GP framework that was implemented on a WSN to make decisions at the node-level. We demonstrated that correlations between sensory-attributes of the node can be computed using GP to perform symbolic-regression at the node resulting in significant saving of power without a significant loss in accuracy. We also demonstrated the convergence of the proposed GP technique and outlined the various issues we faced in developing the GPSense framework.

This being the first-of-its-kind implementation of an in network data mining algorithm, we are hopeful that other in-network data mining algorithms will also be developed. Distributing the entire evolutionary process across various nodes in the network was particularly challenging. The broadcast-collect model of evolution can be improved further using island-models or developing special purpose ADFs which can be shared between various nodes to enhance collaboration.

BIBLIOGRAPHY

[1] Derek Johnson, Ankur M Teredesai and Robert Saltarelli, *Genetic Programming in Wireless Sensor Networks*, European Conference on Genetic Programming EuroGP *2005*, Lussanne, Switzerland, vol. 3447, pp 96-107, April 2005.

[2] J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[3] Riccardo Poli and W.B. Langdon, *Genetic Programming with One-Point Crossover and Point Mutation*, Technical report, CSRP-97-13, University of Birmingham, School of Computer Science, Birmingham, B15 2TT, UK., April 1997

[4] *http://www.talkorigins.org/faqs/genalg/genalg.html#what:selection*

[5] www.cs.binghamton.edu/~kang/teaching/cs580s **brent**-fr.doc

[6] *http://etd.fcla.edu/SF/SFE0000308/CHE_YingZhang_thesis.pdf*

[7] *http://research.cs.ncl.ac.uk/astra/motes.htm*

[8] TinyOS download http://www.tinyos.net/download.html

[9] *http://www.sensorsmag.com/sensors/data/articlestandard//sensors/192006/324975/i2.gif*

[10] Xiaofan Jiang, Joseph Polastre, and David Culler. Perpetual environmentally powered sensor networks. *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on* 15 April 2005, page(s): 463- 468

[11] Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. The MIT Press

[12] Sean Luke and Lee Spector, *A Comparison of Crossover and Mutation in Genetic programming*, Genetic Programming 1997: Proceedings of the Second Annual Conference.

[13] Gianluca Bontempi, Yann-Ael Le Borgne. *An Adaptive modular approach to the mining of sensor network data.* Siam international workshop on data mining in Sensor network, April 21-23, 2005.

[14] Andrea Kulakov, Danco Davcev. *Data mining in wireless sensor networks based on artificial neural-networks algorithms*. Siam international workshop on data mining in Sensor network, April 21-23, 2005.\

[15] Sabine M. McConnell, David B. Skillicorn. *A distributed approach for prediction in sensor networks.* Siam international workshop on data mining in Sensor network, April 21-23, 2005.