Energy-Efficient Sensor Monitoring

Sarvdeep Singh Bindra
Advisor: Dr. Minseok Kwon (jmk@rit.edu)
Rochester Institute of Technology
Email: ssb2339@rit.edu

Abstract—In people-centric sensing, we use mobile nodes (such as smart phones) to collect sensor data. A sensing application distributes sensing tasks to various mobile nodes. The mobile nodes might use in-built sensors or might seek sensors from neighboring mobile nodes or peers to save energy. In this paper, we would be evaluating the approximate distributed algorithms designed for sensor assignment and monitoring. In order to evaluate the algorithms, we process the output of the six algorithms. To accomplish this, we created a library between ruby and Matlab and got results in terms of cost of monitoring of the distributed algorithms.

I. INTRODUCTION

In people-centric sensing, a sensing application generates sensing tasks. These tasks specify what kind of sensor data should be collected and reported back to the application. The sensing application distributes these tasks among a number of mobile nodes. In order to report the sensor data back to the sensing application, the sensors need to be monitored. Continuous monitoring of sensors can drain the battery of the mobile nodes. This can cause the sensing task to become undesirable for the mobile nodes and the mobile nodes might reject the sensing tasks due to the requirement of high energy consumption. The sensing condition is expressed as a boolean expression.

In a decentralized network, the sensing tasks generated by a sensing application are represented in the form of conjunctive normal form (CNF) equations. A CNF equation has multiple sub-clauses. The master assigns the sub-clauses to select helpers. A helper is a mobile node which uses in-built sensors or sensors from neighboring peers to sense the condition and report it back to the master. For example,

\[
\text{CNF clause}(F) = (\text{Windspeed} > 10) \cap (\text{Acceleration} > 30) \cap ((\text{humidity} > 60) \cup (\text{temperature} > 90) \cup (\text{temperature} < 33)).
\]

Represented as: \(x11 \cap x21 \cap (x31 \cup x32 \cup x33)\). Suppose if there are three helpers with sensors: \text{Helper1}: \{\text{Windspeed, Acceleration}\}. \text{Helper2}: \{\text{Humidity, Temperature}\}. \text{Helper3}: \{\text{Acceleration, Humidity, Temperature}\}. One possible assignment can be \(H1 : \{x11, x21\}, H2 : \{x31, x32, x33\}\). Another possible assignment can be \(H1 : \{x11\}, H3 : \{x21, x31, x32, x33\}\).

In this paper, we would be evaluating the performance of the six algorithms developed for assigning the atoms to various helpers. The evaluation is based on cost of monitoring the atoms. The computation of the cost of monitoring depends on the assignment of atoms to the helpers, the cost of the sensors, the probability of the sensors reporting a condition as false, duration of the task and the lambda values of the atoms.

Our evaluations show that, all the algorithms provide comparable results in terms of cost of basic condition monitoring. The cost of deamon condition monitoring is significantly less as compared to the cost of basic condition monitoring.

In the following sections we discuss the related work (Section II) which describes the six algorithms, give an explanation to the implementation (Section III) where we describe the variables on which the computation of the cost of monitoring depends and explain the logic of the data processing. We also explain the role of java as an interface between ruby and Matlab. We present and explain our results (Section IV) as obtained from the Matlab function and conclude (Section VI).

II. RELATED WORK

In [1], the authors have formalized an energy-aware sensor monitoring problem and have proposed a solution for it. The authors have considered \(v\) as the universe of sensors. \(m\) is the master node and \(H\) is the set of helpers. The master node also uses its own sensors, therefore it is a part of the helper set, \(m \in H\). \(O_h\) is the offer set provided by a helper. An offer set determines which conditions belonging to a clause, can a helper sense and report back to the master.

The goal of the problem is to decrease the average energy consumption rate, which is given by the following formula:

\[
C_{\text{rate}} = \sum_{h \in H} \left\{ \left( \sum_{s \in O_h} \tau_{h,s} \rho_{h,s} c_{h,s} \right) + w_h c_h \right\} [1].
\]

In the above formula, \(\tau_{h,s}\) is the sampling rate of helper \(h\) on sensor \(s\), \(\rho_{h,s}\) is the time for which helper \(h\) monitors sensor \(s\) and \(w_h\) is the average transmission bandwidth of helper \(h\). The cost of monitoring \(E[C_{\text{rate}}]\) is computed by assuming that all the sensors are monitored all the time. Therefore, \(\rho_{h,s}\) becomes 1. Therefore, now the helpers report back to the master whenever an atom assigned to a helper changes its value. The cost of monitoring is given by the following formula: \(E[C_{\text{rate}}] = \sum_{h \in H} \sum_{x \in A(h)} \text{Cost}_h(x) [1]\).

DEAMON-ASSIGN [1] assigns the atoms in \(A(F)\), such that the cost represented by the above formula is minimal. The assignment of atoms to helpers plays an important role in deciding the cost of monitoring the atoms at a later stage. In [1], the authors have described two methods of monitoring the sensors. The first one is basic condition monitoring and the other is deamon condition monitoring. Basic condition monitoring monitors all the sensors at a given time. Therefore, it depends on the cost of the sensors. Deamon condition monitoring monitors the smallest set of atoms, which can report the transformation in a clause’s state.

For example, if \(F = (x_1 \cup x_2 \cup x_3) \cap (x_4 \cup x_5 \cup x_6)\), then
deamon monitoring would monitor only \( x_1, x_2 \) and \( x_3 \), since \( x_1 = x_2 = x_3 = false \) would determine the state of the clause, which would be false. Figure 1 shows a possible assignment for the example above.

![Diagram](image.png)

Fig. 1: The clause \( F = (x_1 \cup x_2 \cup x_3) \cap (x_4 \cup x_5 \cup x_6) \) is represented by the above tree with the assignment \( A = \{\{x_1 \cup x_2\}, \{x_3\}, \{x_4 \cup x_5 \cup x_6\}\} \) for helpers \( H_1, H_2 \) and \( H_3 \).

Six approximate distributed algorithms are developed, which can assign the atoms to helpers with minimal cost [2], [3].

**A. Description of Algorithms**

1) **Greedy Algorithm**: The greedy algorithm computes the average cost of helpers (cost of helper/newly covered atoms) at each step. The algorithm selects the helper with the lowest average cost and adds it to the solution [2].

**Algorithm 1 Greedy Algorithm**

1: \( A(F) = \) Atom set of \( F \)
2: \( x = \) set of atoms that helper \( h \) can monitor
3: \( C = \) null, \( H = \) set of helpers
4: \( \text{while} \ A(F) \text{ is not a subset of } C \) \( \text{do} \)
5: \( \text{helper} = \min (\text{cost of helper } i/\text{number of new covered atoms}) \)
6: \( \text{Ch} = \) atoms which helper \( h \) can still cover
7: \( \text{Fh} = \text{collapse(Ch, F)} \)
8: \( H = H - \text{helper} \)
9: \( C = C \cup \text{Ch} \)
10: \( \text{end while} \)
11: Output \( O = \{ \text{Fh: } h \in H \} \)

2) **Better Than Greedy Algorithm**: The better than greedy algorithm is the same as the greedy algorithm, with a slight modification. In better than greedy algorithm we can withdraw previously covered subsets and select new subsets in order to achieve minimum cost. For example, \( H_1 : (1,2,3) \rightarrow \text{cost} = 12, H_2 : (4,5,6) \rightarrow \text{cost} = 12, H_3 : (1,2,3,4,5,6) \rightarrow \text{cost} = 18 \). The greedy algorithm will pick up \( H_1 \) and \( H_2 \) as the solution in order to cover all six sensors and will end up with a cost of 24. Better than greedy algorithm will pick up \( H_1 \), will check the other helpers for their cost and then withdraw \( H_1 \) from the solution. It will select \( H_3 \) as the solution for covering all the 6 sensors and will end up with a cost of 18 [2].

3) **Tabu Search Algorithm**: Tabu search algorithm keeps track of its previous states. It moves to the next state even if the state becomes worse [2]. The flowchart for the tabu search algorithm is as represented in figure 2.

![Flowchart](image.png)

Fig. 2: Flowchart for Tabu Search

4) **Best Local Search**: The best local search algorithm is similar to simulated annealing with a minor modification. Simulated annealing picks up a neighbor even if it has a higher cost with a probability. Best local search stops when it encounters a neighbor with a higher cost [3]. The pseudocode for best local search is as follows: [3].

**Algorithm 2 Best Local Search Algorithm**

1: \( \text{findBestSolution (uniqueSensorsInExp, currentSolution)} \)
2: \( \text{costSolution} = \) cost of current solution
3: \( \text{while a solution with worse cost is encountered do} \)
4: \( \text{listOfNeighbors} = \text{replace each helper in current solution with every other helper} \)
5: \( \text{bNeighbor} = \) neighbor with lowest cost in \( \text{listOfNeighbors} \)
6: \( \text{costBNeighbor} = \) cost of \( \text{bNeighbor} \)
7: \( \text{if (costBNeighbor < cost of solution)} \) \( \text{then} \)
8: \( \text{currentSolution} = \text{bNeighbor} \)
9: \( \text{else break} \)
10: \( \text{end if} \)
11: \( \text{end while} \)
12: \( \text{return currentSolution} \)

5) **Simulated Annealing Algorithm**: The simulated annealing algorithm generates an initial solution. This initial solution covers all the sensors or atoms in the given clause. With a high ‘temperature’ at the start, which decreases to 0 gradually, a valid neighbor is found for the solution. If the cost of the valid neighbor is less than the cost of the current solution then this valid neighbor is included in the solution. If the cost of the valid neighbor is not lower than the cost
of current solution, it is still included in the solution with a probability, which is proportional to the temperature which decreases gradually [3].

The flowchart for simulated annealing is as represented in figure 3:

![Flowchart for Simulated Annealing](image)

**Fig. 3: Flowchart for Simulated Annealing**

6) **Multiple Start Best Local Search**: Multiple start best local search is the same as best local search. It starts with multiple initial solutions and converges to one final solution [3].

III. IMPLEMENTATION

The approximate distributed algorithms are implemented in ruby and the cost computing function is implemented in Matlab. We created a library to connect the distributed algorithms in ruby to the cost computing function in Matlab. Java acts as an interface between ruby and Matlab. It is responsible for the connection between ruby and Matlab and also the conversion of data into appropriate format for Matlab and for ruby. We processed the output of the distributed algorithms which provide the assignment of helpers, created the input variables for the cost computing function in ruby and achieved the results for the cost of monitoring.

A. **Input variables for Matlab**

1) **Tau**: Tau is the task.tau value, which is taken as 0.5 because it is considered as this particular value in anonymous-lib.rb

2) **Duration**: This is the task duration. Its value is 10000 since this value is used by deamon.rb and this is the value used in the sample input for the Matlab function.

3) **Lmd**: Lmd is the array of lambda values for every atom. The value for each atom is calculated using the following formula.

\[
\text{Interval} = \min_{\text{interval}} + \text{random}(\max_{\text{interval}} - \min_{\text{interval}})
\]

Lambda value = 1/interval.

Here the value of \(\min_{\text{interval}}\) is 60.0 and the value of \(\max_{\text{interval}}\) is 100.0.

4) **P**: Every atom has a probability of being false. P is the array of probability values for all the atoms. The value for each atom is computed as follows

\[
p = \min_{\text{prob}} + \text{random}(\max_{\text{prob}} - \min_{\text{prob}})
\]

Here the value of \(\min_{\text{prob}}\) is 0.5 and the value of \(\max_{\text{prob}}\) is 0.6.

5) **C**: It is an array of cost read for every atom. The value for each atom is computed using the following formula:

\[
c = \min_{\text{cost}} + \text{random}(\max_{\text{cost}} - \min_{\text{cost}})
\]

Here the value of \(\min_{\text{cost}}\) is 0.5 and the value of \(\max_{\text{cost}}\) is 0.6.

6) **Cntfy**: It is an array consisting of cost of notification for every atom. The cost of notification for every atom is considered as 0.

7) **F**: F is a list expression of the boolean function. In short, F is a matrix way of describing the given boolean expression in CNF and the assignment at the same time.

From F, we can get the number of atoms. In the example, there are four atoms, say \(x_1, x_2, x_3, x_4\). The number of row in F is the number of clauses. So, if F has three rows, the boolean expression is in the form of \((\ldots) \cap (\ldots) \cap (\ldots)\). We need to define how many atoms each clause contains.

Suppose first clause contains one, second clause contains two, third clause contains one atom. Then, the formula turns out to be \((x_1) \cap (x_2 \cup x_3) \cap (x_4)\).

Therefore we might define \(F=\{1; 2 3; 4\}\). However, on top of this information, we have to encode the assignment to helpers. That is, which helper is in charge of which atoms. One possible assignment is to assign \(x_1\) to first helper, \(x_2\) and \(x_3\) to second helper, and \(x_4\) to the third helper. The matrix below represents the above assignment.

\[
F = \begin{bmatrix}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

The first row \([1, 1, 1]\) indicates that [(there is one helper for this clause), (this helper monitors one atom), (the atom’s number is 1)]. The second row \([1, 2, 2, 3]\) indicates that [(this clause has one helper), (this helper monitors two atoms), (first atom’s number is 2), (second atom’s number is 3)]. The third row \([1, 4, 1]\) indicates that [(this clause has one helper), (this helper monitors one atom), (this atom’s number is 4)].
B. Ruby and Java implementation

The processing of the output of the six algorithms and the generation of the input variables for the matlab function is implemented in ruby. The ruby code reads the output of each algorithm, counts the number of helpers in the output, which determines the number of clauses and the number of rows in the matrix \( f \text{array} \). For every helper, the ruby code reads the number of atoms, reads the atom or sensor list which is covered by that particular helper and sets the corresponding row in \( f \text{array} \) for that helper. It is also responsible for generating the array with probability values for every atom, the array consisting of lambda values of every atom, setting the tau and duration values and generation of the array consisting of costs for every atom.

The java code depends on the matlabcontrol-4.1.0.jar library to connect to Matlab. This library sends the data to Matlab in the form of objects. Therefore, all the data passed from the ruby code has to be converted into objects. The java code is responsible for converting the input variables obtained from ruby into objects. It then creates an object array, which is filled with the input variables converted to objects. This object array acts as the argument list for the Matlab function. Once the Matlab function finishes the computation, it returns the results in the form of an object array. This is because every variable in Matlab is considered as an array. Therefore, the java code converts the variables in the object array to double data type. After converting the data to the appropriate data type, the java code returns the result to the ruby code in the form of an array filled with the result values. When the ruby code receives the array filled with the result values, it traverses the array and prints the output to the console. The pseudocode for the processing of output and generation of input variables is explained in Algorithm 3 and the working of the java code is explained by the flowchart in figure 4.

![Fig. 4: Working of the Java code](image)

C. Steps for Execution

1) Execute each algorithm individually to get the results from Matlab.
2) Use the jruby command to execute the ruby code.

If the environment and all the tools are set up correctly then each algorithm will successfully call the Matlab function and get the result from the function.

IV. RESULTS

We present the results of experiments with a various number of helpers and sensors. The number of sensors to be covered in the CNF is twice that of the number of helpers. Each helper has 3 to 8 sensors chosen at random. This facilitates a subset of helpers to cover all the sensors in the CNF with the lowest cost possible.

The assignment of atoms to helpers plays an important role in the computation of the cost of monitoring. The reason is that if the assignment of the atoms to helpers is efficient, the algorithm will have a low cost of monitoring. The cost of monitoring also depends on the cost of notification for every atom. A helper notifies the master whenever an atom, monitored by that helper changes its state. For the purposes of the experiments, the cost of notification was considered as zero. We present the results obtained for basic condition monitoring, which monitors all the atoms and the results of deamon condition monitoring, which considers the assignment of the atoms to the helpers.
A. Cost of Basic Condition Monitoring

<table>
<thead>
<tr>
<th>Number of Helpers</th>
<th>Greedy Algorithm</th>
<th>Simulated Annealing</th>
<th>Better than</th>
<th>Best Local Search</th>
<th>Tabu Search</th>
<th>Multiple Start</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>565393.07</td>
<td>54546.31</td>
<td>55939.56</td>
<td>55797.25</td>
<td>54609.01</td>
<td>54275.52</td>
</tr>
<tr>
<td>20</td>
<td>110443</td>
<td>110612.03</td>
<td>110038.98</td>
<td>108587.61</td>
<td>109998.28</td>
<td>110919.66</td>
</tr>
<tr>
<td>30</td>
<td>166944.85</td>
<td>165304.83</td>
<td>165455.3</td>
<td>164452.43</td>
<td>16471.16</td>
<td>164418.33</td>
</tr>
<tr>
<td>40</td>
<td>218988.27</td>
<td>219161.97</td>
<td>219001.12</td>
<td>219867.57</td>
<td>220078.77</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>274743.05</td>
<td>275696.98</td>
<td>275789.56</td>
<td>276144.55</td>
<td>276587.03</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>328769.1</td>
<td>331530.26</td>
<td>332547.6</td>
<td>331535.31</td>
<td>330858.49</td>
<td>331314.68</td>
</tr>
</tbody>
</table>

TABLE I: Cost of Basic Condition Monitoring

Basic condition monitoring considers the cost of every atom in the clause, the duration of the task and the tau value.

Fig. 5: Variation in Cost of Basic Condition Monitoring

Figure 5 indicates, that the cost of monitoring for every algorithm increases linearly as the number of helpers are increased. This is because, basic condition monitoring does not consider the assignment of atoms to the helpers. Therefore, the cost of monitoring of all the algorithms are comparable.

Fig. 6: Performance of the algorithms

Figure 6 indicates, that there is a minor difference in the cost of monitoring of all the algorithms. Although the difference between the cost of monitoring is minor, the above graph indicates that greedy algorithm provides the lowest cost compared to other algorithms.

B. Cost of Deamon Condition Monitoring

The cost of deamon condition monitoring is based on the assignment of the helpers for the input clause, the cost of every atom or sensor which is present in the final assignment, the tau value, the duration of the task, lambda values of all the atoms or sensors in the final assignment and on the probability of every atom or sensor reporting the condition as false.

<table>
<thead>
<tr>
<th>Number of Helpers</th>
<th>Greedy Algorithm</th>
<th>Simulated Annealing</th>
<th>Better than Greedy</th>
<th>Best Local Search</th>
<th>Tabu Search</th>
<th>Multiple Start</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7487.7</td>
<td>16510.88</td>
<td>14535.89</td>
<td>16014.93</td>
<td>16082.26</td>
<td>11617.99</td>
</tr>
<tr>
<td>20</td>
<td>10538.23</td>
<td>21635.09</td>
<td>24509.37</td>
<td>22769.15</td>
<td>24210.4</td>
<td>23044.79</td>
</tr>
<tr>
<td>30</td>
<td>5447.8</td>
<td>27353</td>
<td>26694.8</td>
<td>23957.2</td>
<td>27123.04</td>
<td>30603.17</td>
</tr>
<tr>
<td>40</td>
<td>4838.48</td>
<td>25295.09</td>
<td>27367.8</td>
<td>30489.13</td>
<td>24871.56</td>
<td>27382.86</td>
</tr>
<tr>
<td>50</td>
<td>4818.58</td>
<td>30398.52</td>
<td>29675.3</td>
<td>28042.79</td>
<td>32504.62</td>
<td>22445.46</td>
</tr>
<tr>
<td>60</td>
<td>5915.57</td>
<td>26622.92</td>
<td>27486.3</td>
<td>26976.54</td>
<td>21702.41</td>
<td>24790.51</td>
</tr>
</tbody>
</table>

TABLE II: Cost of Deamon Condition Monitoring

Fig. 7: Variation in Cost of Deamon Condition Monitoring

Figure 7 indicates, that the greedy algorithm shows a slight variation in the cost of deamon monitoring initially but as the number of helpers increases it shows near constant result. The above graph also indicates that the other algorithms provide results in approximately the same range as the number of helpers increase.

Fig. 8: Performance of the algorithms

Figure 8 indicates, that the greedy algorithm provides the lowest result in terms of cost of deamon condition monitoring. This is because greedy algorithm provides a better assignment than the other algorithms. It also indicates that the cost of deamon condition monitoring for the greedy algorithm does not vary much as the number of helpers is increased. Except for greedy algorithm, all the other algorithms provide comparable results.
V. ACKNOWLEDGEMENTS

I would like to thank Dr. Minseok Kwon for his guidance and for providing me an opportunity to work under him. It was a great experience to work on an on-going research project. The work on this capstone project was carried out at Rochester Institute of Technology.

VI. CONCLUSION

Greedy algorithm provides the best results in terms of cost of both basic condition monitoring and daemon condition monitoring. Simulated Annealing and Multiple Start Best Local Search provide the second best result in terms of cost of both forms of monitoring.

REFERENCES

