Distributed Monitoring in 6LoWPAN based Internet of Things

B. Mostafa[1,2,3], A. Benslimane[2], E. Boureau[1] , M. Molnar[1], and M. Saleh[3]

1. LIRMM, University of Montpellier, 161, rue Ada 34095 Montpellier Cedex 5, France
2. CRI, French University in Egypt, B.P 21 - El Shorouk City, 11837, Egypt
3. Faculty of Computers and Information. Cairo University, 5 Dr. Ahmed Zewail Street, Postal Code: 12613, Orman, Giza, Egypt

Abstract—The emergence of the Internet of Things (IoT) is introducing more and more services and applications such as smart cities. For some services, the availability of elements and the connectivity between them are necessary. The robust functioning of a fragile and often dynamic system in IoT needs strong monitoring which is investigated in this paper. IPv6 Routing Protocol for Low Power and Lossy Networks (RPL) is the standardized routing protocol over IP-connected IoT networks. In tandem with RPL, we propose a new solution which aims to achieve distributed monitoring with minimal computational complexity. The main objective is to increase robustness in IoT via monitoring the links in the Destination Oriented Directed Acyclic Graphs (DODAG) constructed by RPL. Although RPL has a reactive repairing mechanism to tackle robustness in connectivity in case of node failures, we found that for real-time critical applications it is important to use proactive approaches for preventing faults and making recovery for connectivity faster. The problem can be modeled as a Vertex Cover Problem (VCP) on the DODAG. Such problem is a well known NP-hard optimization problem. The monitoring should be simple and energy aware. We demonstrate that the monitor placement in our case is only Fixed-Parameter Tractable, and also polynomial-time solvable, which is the best case.

Keywords—IoT; RPL; link monitoring; Vertex Cover.

I. Introduction

The 6LowPAN based Internet of Things (IoT) networks tend to experience unexpected communication problems during deployment, because resource-constrained embedded devices are unreliable by nature for a variety of reasons, such as uncertain radio connectivity and battery drain [1]. To that end, monitoring techniques for detecting, localizing and remedying network failures in IoT will definitely develop in significance. The primary objective of network monitoring in general is mapping the symptoms of the detected problems to possible root causes to take corrective measures [2].

Low power and Lossy Networks (LLN) experience several communication challenges such as energy and computational limitations of devices and extensive protocol overheads against memory. To handle such challenges, the Internet Engineering Task Force (IETF) has standardized the Routing Protocol for Low Power and Lossy Networks (RPL) as the routing protocol for IP-connected IoT [3] [4]. RPL is a self-healing routing and topology control protocol as it is able to respond to some node or link failures. It uses reactive local and global repair approaches for network recovery. However, the recovery time, which is the time required to establish a new route could be critical. If for any reason, the links connecting the node with its neighbors are all broken, as in the case of a sudden failure of multiple links, the node will not be able to regain connectivity quickly.

RPL favors the use of reactive repair approaches as they are more energy-efficient; to minimize the cost of monitoring links that are not being used [4]. Furthermore, the neighbor unreachability detection (NUD) is not mandatory in RPL and active mechanisms for probing neighbors regularly do not exist in ContikiRPL implementation [5] [6].

Although a significant number of IoT applications are not time-sensitive, there is a whole class of real-time, mission-critical applications; where data must be processed and shared instantly and within strict reliability constraints. For instance, critical control and fault detection applications; where corrective actions must be taken with little delay.

For critical, real-time IoT applications, instead of the embedded RPL reactive repair approaches, proactive monitoring mechanisms should be used for early detection and prevention of network failures. Node and link failures could then be detected beforehand, as a kind of preventive maintenance. This could greatly improve the robustness in connectivity, reliability and eventually Quality of Service (QoS) in the network, which will significantly increase the uptake of the technology by stakeholders. The added cost for link monitoring could be tolerated for critical-time IoT applications as it will help accelerate recovery, and decrease node unreachability times.

Consequently, we were motivated to investigate the problem of proactive link monitoring for IoT. The main objective is to increase robustness in IoT via monitoring the links in the Destination Oriented Directed Acyclic Graphs (DODAG) constructed by RPL. Distributed monitoring can be achieved by placing multiple monitoring nodes on the network. Those nodes are responsible for anticipating node/link failures by continuously tracking the status of the links in the DODAG and taking the corrective actions before failure happens. In this paper, we focus on calculating the

978-1-5090-1743-0/16/$31.00 ©2016 IEEE

minimum number of monitoring nodes required to track the entire set of links. We modeled the problem as a Vertex Cover Problem (VCP) on the DODAG. VCP is NP-hard, which implies that unless P = NP, efficient algorithms for solving it don't exist [13]. However, due to the battery and computational constraints of embedded devices, the monitoring should be simple and energy-aware. We propose an algorithm to convert a DODAG into a nice-tree decomposition which makes solving the generally NP-hard VCP on this special graph achieved in polynomial-time.

The rest of the paper is organized as follows: section 2 is a brief overview of the RPL topology construction, repair mechanisms and a background of the concepts of vertex cover and tree decomposition problems. Section 3 presents the monitoring requirements, the proposed algorithm and proof of termination, followed by an analysis of the algorithm in Section 4. The conclusions and future research are presented in Section 5.

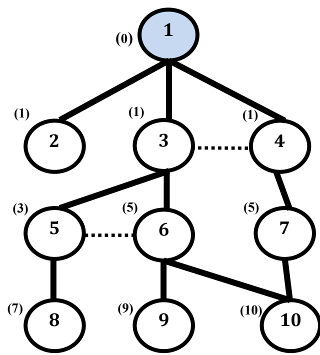

Fig. 1. A sample RPL DODAG that has N nodes. The root is shaded, the numbers adjacent to the circles are the ranks, and the routes through siblings are dahsed.

II. Background

A. RPL DODAG

RPL is a versatile protocol that supports different modes of operation: many-to-one communication from the constrained nodes towards the root (typically the 6LoWPAN border router 6BR), one-to-many communication from the root to the constrained nodes (constrained in terms of battery and memory, etc.) and, one-to-one communication between the constrained nodes. The basic component of RPL is a Destination Oriented Directed Acyclic Graph (DODAG), a sample is shown in Fig. 1. It is a hierarchical organization, where each node has a node ID, one or more parents (except for the DODAG root), and a list of neighbors [7].

Nodes also have a rank that determines their individual position in the hierarchy with respect to the DODAG root and relative to other nodes, which are the numbers adjacent to the nodes in Fig.1. In RPL, the rank as well as the parent selection is calculated via an Objective Function (OF) that might use factors such as link quality, node energy and link reliability in its calculation [5].

The DODAG construction depends on the Neighbor Discovery Protocol (NDP) and Upward route discovery protocol in 6LoWPAN. They allow a node to join a DODAG by discovering neighbors that are members of the DODAG and identifying a set of parents [4]. The protocols build three logical sets of link-local nodes. Starting by the candidate neighbor set, which is a subset of the nodes that can be reached through link-local multicast. Then, the parent set, which is a restricted subset of the candidate neighbor set. Finally, the preferred parent set, a member (or members) of the parent set which is the preferred next hop in Upward routes. The exact policies for selecting neighbors is implementation dependent and driven by the OF.

B. DODAG Repair Mechanisms

Repair mechansims are of foremost significance for a routing protocol to update routes dynamically and adapt the network topology to potential failures. For that purpose, RPL supports two integral repair mechanisms, in particular local repair and global repair [6].

When a node detects a network failure (e.g. a link between two nodes fails), it triggers local repair (see Fig. 2). It consists in searching for a backup path urgently without attempting to repair the entire DODAG. However, this alternate recovery path may not be the best path with respect to the defined OF.

The second mechanism is the global repair; it prompts the fundamental reconstruction of the entire network topology, but on the expense of additional control traffic in the network [1]. Nodes in the new DODAG version can choose a new position whose rank is not dependent on the previous one. The global repair is considered as an entire re-optimization of the routes. However, there are times when the nodes are unreachable during the DODAG rebuilding process. Therefore, relying solely on global repairs is neither efficient, in terms of the number of control messages, nor reliable especially for critical-time IoT applications where there is no tolerance for nodes unreachability.

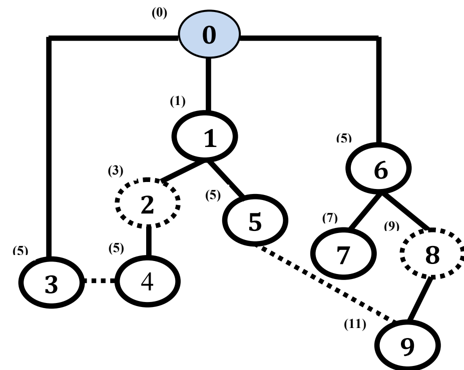

Fig. 2. A DODAG after the local repairs. Borders of inactive nodes are dotted, repaired routes are dashed, the root is shaded, and the numbers adjacent to the circles are the ranks [12].

C. Vertex Cover & Tree Decompositions

The Vertex Cover Problem (VCP) is a well known NP-hard graph optimization problem. VCP is defined over an undirected graph $G = (V, E)$ and searches for a set of vertices

$S \subseteq V$ such that for each edge $e \in E$ at least one of its endpoints belongs to S and |S| is as small as possible.

There are several general approaches for attacking NP-hard problems, among them approximation algorithms, fixed-parameter algorithms, and heuristics. VCP is one of the best studied problems concerning fixed-parameter tractability [8]. Several techniques in parameterized complexity were successfully applied to VCP, as for instance data reduction, depth-bounded search trees, and dynamic programming [9].

Another approach for solving NP-hard graph optimization problems is the concept of tree decompositions for graphs, which was introduced by Robertson and Seymour [10] and plays an important role in algorithmic graph theory.

Tree decompositions were motivated by the observation that many NP-complete problems are easy to solve on trees, connected graphs without cycles [5]. Trees are restricted structures when compared with general graphs; [10] addressed how hard problems can be solved for graphs that are "like" trees. Tree decompositions are the formal way to describe the "tree-likeness" of a given graph [8]. See Fig. 3 for an example of tree decompositions.

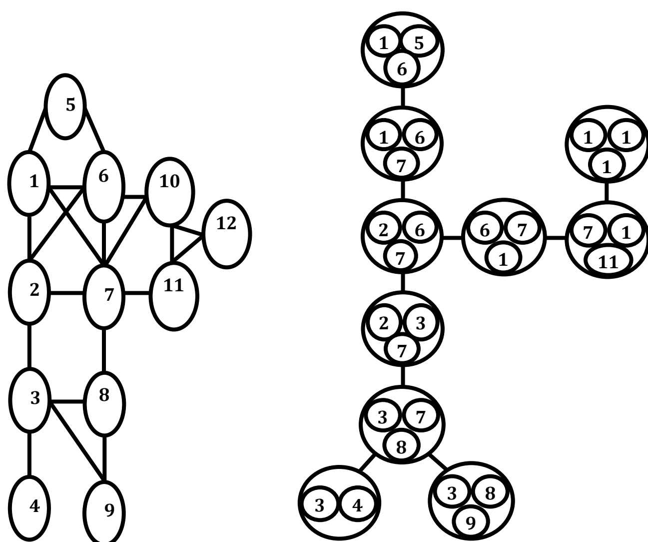

Fig. 3. Example of a graph G and its corresponding tree decomposition T [8].

In the following definitions we give a brief description of tree decomposition, treewidth, and nice-tree decomposition. Fig. 3 illustrates an example of an optimal tree decomposition T. T is optimal in the sense that there is no tree decomposition for the given graph G such that every bag contains fewer than three vertices. Observe that the properties of tree decompositions as stated in Definition 1 hold [5].

The concept of treewidth has been proven to be very useful in algorithmic graph theory. Several problems that are NP-hard on general graphs are polynomial-time (some even linear-time) solvable on graphs with bounded treewidth. These problems include graph coloring and vertex cover [3].

The usual approach of tree decomposition based algorithms is dynamic programming. The fact that the running time of dynamic programming algorithms on tree decompositions is commonly of the complexity $O(c^k * P(n))$, where k is the width of the tree decomposition, $P(n)$ is a polynomial function of n, and n is the size of the instance, is useful for our problem. As these algorithms can be executed in a reasonable amount of time on graphs where k is relatively small (less than 20, for example) [11]. If we restrict graph problems to graphs with small tree decompositions, we can solve many NP-hard problems by using dynamic programming. Such problems are called Fixed-Parameter Tractable (FPT) [3] because they have algorithms that run in polynomial-time when the parameter, namely treewidth of the tree decomposition is fixed. Furthermore, algorithmic solutions are often easier when working with a nice-tree decomposition (cf. Definition 3) instead of a tree decomposition[11].

Definition 1: Tree Decomposition [8]

Let G = (V, E) be a graph. A tree decomposition of G is a pair $(\{X_i: i \in I\}, T)$, where each $X_i \subseteq V$ is called a bag, and T is a tree with the elements of I as nodes. The following three properties must hold:

1. $\bigcup_{i \in I} X_i = V$,
2. for every edge $e \in E$ there exists a bag X_i with $i \in I$ and $e \subseteq X_i$, and
3. for all $i, j, k \in I$, if j lies on the path from i to k in T then $X_i \cap X_k \subseteq X_j$.

The third property is equivalent to the requirement that, for each $v \in V$, the nodes of all bags containing v induce a subtree of T.

Definition 2: Treewidth [8]

The width of a tree decomposition is the maximum of $|I(v)| - 1$ over all $v \in V_T$.

Definition 3: Nice-Tree Decomposition [8]

A tree decomposition $(\{X_i: i \in I\}, T)$ is nice if:

- T is rooted at a designated node $r \in I$, called root node,
- every vertex of T has at most two children (i.e. the tree is a binary tree),
- if i is a leaf of T, $|I(i)| = 1$ ($I(i)$ is called a start bag),
- if i has two children j and k, $I(i) = I(j) = I(k)$ (in this case $I(i)$ is called a join bag), and
- if i has one child j, then either:
  - $|I(i)| = |I(j)| - 1$ and $I(i) \subset I(j)$, (in this case $I(i)$ is called a forget bag), or
  - $|I(i)| = |I(j)| + 1$ and $I(j) \subset I(i)$ (in this case $I(i)$ is called an introduce bag).

Based on this information, converting a DODAG into a nice-tree decomposition with bounded treewidth will yield the advantage of solving many NP-hard graph optimization problems in polynomial-time. This approach can be used in our monitoring problem, which we addressed in this paper.

III. Converting a DODAG into Nice-Tree Decomposition with Unity Treewidth

A. IoT Link Monitoring Requirements

For critical-time IoT applications in wireless networks, it is crucial to keep track of all the links in the network, especially the inactive ones which are not currently used for routing.

In order to meet the reliability requirements for critical applications, we were motivated to study the problem of network monitoring for IoT through monitoring the links in DODAGs. We propose a distributed monitoring model where there are several nodes carefully placed on the DODAG, which are responsible detecting and localizing network failures.

One of the main challenges for network monitoring is determining where to embed the monitoring nodes. Moreover, the monitoring computational cost, battery and memory requirements should be minimal in order to satisfy the low cost and energy constraints of IoT networks. Therefore, it is important to minimize the number of monitoring nodes placed on the graph, meanwhile balancing the monitoring load to maintain the status of all links and also maximize network lifetime. In other words, the objective is minimizing the number of monitoring nodes that "cover" the entire graph.

Taking the stated requirements into consideration, finding out the minimum number of monitoring nodes placed on the graph to keep track of all the links in the network can be modeled as the classic VCP. However, as mentioned in Section 2.3, VCP is NP-hard for general graphs and FPT when solved by dynamic programming on tree decompositions with bounded treewidth.

In view of this information, converting a DODAG into a nice-tree decomposition with bounded treewidth will yield the advantage of solving the generally NP-hard graph monitor placement problem in this special graph in polynomial-time. To the best of our knowledge, there is no work that tended to this problem in the literature.

B. Proposed Algorithm

A DODAG consists of a set of vertices v_j: $v_j \in V$, where j is a unique identifier for each vertex in D, and a set of edges E. Each vertex in the DODAG is connected via an edge $e_j \in E$ to one of three types of nodes: parents $P(v_j)$ (preferred and alternative), children $C(v_j)$ or siblings $S(v_j)$.

The objective of the proposed algorithm (**Algorithm 1**) is to convert any DODAG D, into a nice-tree decomposition with unity treewidth, while holding the properties of nice-tree decompositions in Definition 3. Bounding the treewidth to the value of one will have the effect of reducing the complexity of solving the VCP on the DODAG to be polynomial-time, which is the main contribution of this paper.

Algorithm 1: Convert DODAG into Nice-Tree with Treewidth 1

Input: DODAG $D = (\{v_j: v_j \in V\}, E)$,

Output: Nice Tree Decomposition $(\{X_i: i \in I\}, T)$ with unity Treewidth

Let $c_r \in C(v_j)$ where $C(v_j)$ is the set of children of vertex v_j in DODAG D

Let $s_r \in S(v_j)$ where $S(v_j)$ is the set of siblings connected to vertex v_j in DODAG D

Let the set of Non-Leaf nodes in D be $\{NL: NL = \bigcup v_j \in D \text{ where } |C(v_j)| + |S(v_j)| > 0\}$

Step 1 Initialization

Step 1.1 Initialize an empty tree T then set its root to a bag that only includes the DODAG root

Step 1.2 Sort ascendingly NL according to the ranks of v_j

Step 2 **While** $NL \neq \emptyset$ **do**

Step 2.1 Select v_j from NL {its top vertex}

Step 2.2 Search T for the first bag $X_i = \{v_j\}$ using breadth first search

Step 2.3 Set $L = C(v_j) \cup S(v_j)$ where $s_r \in NL$

Step 2.4 Let no_of_ required_leaves = $|L|$

Step 2.5 **If** (no_of_ required_leaves > 1)

then $t =$ ConstructBinaryTree (X_i, no_of_ required_leaves)

At X_i augment T with t

End If

Step 2.6 **While** $l \leq$ the no_of_ required_leaves

Step 2.6.1 Make leaf$_l$ a 'forget bag' via branching out one child X_k, where $X_k = \{v_j, v_q\}$ and $v_q \in L$

Step 2.6.2 Make X_k an 'introduce bag' via branching out one child $\{v_q\}$

Step 2.6.3 $L = L \setminus v_q$

Step 2.6.4 $l = l + 1$

END While

Step 2.7 $NL = NL \setminus v_j$

END While

END

Algorithm 2: ConstructBinaryTree

Input: Bag B and Z = required number of leaves

Output: A binary tree with the required number of leaves and all its bags are equal to B

Step 1: Create two branches; where each branch has a bag $= B$

Step 2: **If** ($Z > 2$)

then t = **ConstructBinaryTree** (B, $Z - 1$)

Augment the bag at the right branch with t

End If

END

TABLE I. ALGORITHM 1 VARIABLES

Variable	Description
D	The Destination Oriented Directed Acyclic Graph composed of a set of vertices $v_j \in$ V and a set of edges $e_j \in$ E
$C(v_j)$	The entire set of children of vertex v_j
$S(v_j)$.	The entire set of siblings of vertex v_j
c_r	A child of vertex v_j
s_r	A sibling of vertex v_j
X_i	A bag containing subset of V
I	The entire set of bags in the tree decomposition
T	A tree with the elements of I as nodes
NL	The set of Non-Leaf nodes in the DODAG
no_of_required_ leaves	The number of required leaves for v_j in the tree decomposition.

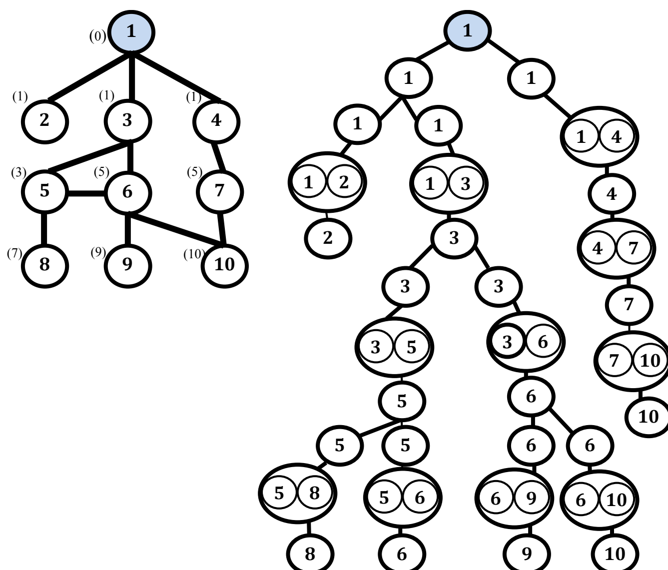

Fig. 4 (a). DODAG D.

Fig. 4 (b). Nice-tree decomposition of DODAG D with unity treewidth.

The algorithm is divided mainly into an initialization step (Step 1) and a main loop (Step 2). TABLE I summarizes the variables in **Algorithm 1**. Step 1.1 initializes a binary tree T and sets its root with a bag that only contains the DODAG root, which is considered as the destination of any node in D. In Step 1.2 we take all the Non-Leaf nodes in the DODAG, store them in a vector *(NL),* and then sort them ascendingly according to their ranks in *D*. The main part of the algorithm is implemented in Step 2, which iterates over the vector *NL* and constructs a binary tree for each of those vertices; by calling the recursive subroutine **ConstructBinaryTree** (see **Algorithm 2**).

Taking the first vertex in *NL,* steps 2.3 and 2.4 determine the required number of leaves (no_of_required_leaves) associated with it, which is the number of its children plus the number of siblings connected to it that are not already included in the tree decomposition.

The recursive subroutine **ConstructBinaryTree** is called at step 2.5, which takes as input a bag *B* and the required number of leaves and returns a binary tree *t* with the required number of leaves and all its bags are set equal to the input bag *B*. The returned tree *t* is then augmented with *T* at the right location; the location of the bag at current iteration.

Step 2.6, is an inner loop for all the leaves (the vertex's children or siblings that are not already included in *T*). In step 2.6.1, we make the leaf a 'forget bag' via branching out one child bag that contains two elements: the leaf and one of the vertex's children (or siblings), in list *L*. Then in step 2.6.2, we make this child bag an 'introduce bag' via branching out a child bag that contains only one element; i.e. the vertex's child or sibling.

Finally, in step 2.6.3, we remove this vertex's child or sibling from the *L* list. After the algorithm exits this inner loop, the vertex is then removed from the vector *NL* at step 2.7; and the outer loop continues until there are no more vertices left in *NL*. Fig. 4(a) shows a DODAG *D* and Fig. 4(b) shows the output of **Algorithm 1**, which is the nice-tree *T* of *D* with unity treewidth.

C. Proof of Termination

Lemma 1: The conversion of a DODAG (**Algorithm 1**) into a Nice-Tree terminates.

Proof: In order to prove that **Algorithm 1** terminates we need to show that the two nested loops, at steps 2 and 2.6 eventually terminate. At Step 2 the While loop starts with a finite subset of vertices, namely *NL*. The cardinality of *NL* is altered only at step 2.7, where it decreases by one. Since *NL* is finite and is strictly decreasing in cardinality with each iteration of the loop, the loop condition ($NL \neq \emptyset$) is eventually falsified and therefore the loop terminates.

At step 2.6, the second loop iterates until *l* is greater than a finite integer quantity, the no_of_required_leaves. Note that *l* initially has the value of one, and the value of the no_of_required_leaves is the cardinality of the finite subset of vertices *L*. The quantity (no_of_required_leaves – *l)* is an integer that strictly decreases with each iteration of the loop. Therefore, eventually (no_of_required_leaves – *l)* is a negative quantity, thus (no_of_required_leaves > *l*), and the loop terminates.

Lemma 2: The construction of a binary tree (**Algorithm 2**) terminates.

Proof: Since **Algorithm 2** is a recursive function, to prove that it terminates we need to show that its arguments get strictly smaller whenever there is a recursive call. The argument *Z* which is the required number of leaves is a finite

integer that starts with a value greater than one, then strictly decreases until it reaches the value two, where the condition ($Z > 2$) is falsified and the recursive function terminates.

IV. Analysis

A. Complexity Analysis

In any nice-tree decomposition produced by **Algorithm 1**, there are two types of bags in the nice-tree namely:

- single-element bags, which contain one vertex of the DODAG, and
- two-elements bags, which contain two vertices that are connected via an edge in the DODAG.

The number of unique single-element bags is equal to the number of vertices in the DODAG. In addition, any two-elements bag will always be a unique bag; as the algorithm is designed to only map once an edge into a corresponding two-elements bag. Hence, we can conclude that the number of two-element bags is equal to the number of edges in the DODAG. Therefore, the number of unique bags in the nice-tree decomposition is simply the summation of the number of vertices and the number of edges in the DODAG.

Single-element bags are either Non-Leaf single-element bags or leaf single-element bags. By experimental analysis, we identified the number of occurrences of any Non-Leaf single-element bag as {2*(no of required leaves) – 1}. Moreover, the number of occurrences of any leaf single-element bag is determined by the number of alternative parents and siblings connected to its corresponding vertex, in the DODAG. Hence, we can conclude that the number of occurrences for both leaf and Non-Leaf single-element bags is of the order of $O(v)$ at maximum, where v is the number of vertices. This leads to the conclusion that the total number of single-element bags is at maximum v^2.

The complexity of the proposed algorithm is measured by the number of bags generated for each DODAG to be a nice-tree decomposition. Therefore, we can conclude that the proposed algorithm has a complexity of $O(m + v^2)$, where m is the number of edges in the DODAG, which is polynomial in time.

Moreover, as mentioned before in Section 3, the running time of dynamic programming algorithms on tree decompositions is typically of the form $O(c^k * P(n))$, where k is the treewidth and n is the size of the instance. Using the proposed algorithm for the construction of the nice-tree decomposition from DODAGs, we will have the treewidth to be only one. Consequently, our particular VCP turns out to be just polynomial in time.

B. Discussion

The initial objective was to monitor the unreliable links in a 6LowPAN network for IoT. Distributed monitoring is achieved via placing several monitoring nodes on the network, which track the status of the links in their neighborhood. We modeled the problem as the classic VCP, which finds out the minimum number of nodes that cover all links. For the purpose of reducing the complexity of solving such NP-hard problem, we proposed converting the DODAG into a nice-tree decomposition with unity treewidth. In order to verify that modeling the problem in such a way actually reaches the initial objective stated above, we need to prove that the DODAG maps the entire network, i.e. it contains a vertex and an edge for every node and link in the 6LoWPAN network.

As mentioned in Section 2 the DODAG construction in RPL is based on the NDP in 6LoWPAN which provides the candidate neighbor set for each node, but the exact policies for selecting neighbors and parents is implementation dependent and driven by the OF [4].

Let the candidate neighbor set be $N(v_j)$, and in **Algorithm 1** we defined the set of parents, children and siblings of each vertex as $P(v_j)$, $C(v_j)$ and $S(v_j)$, respectively. If we design the OF to construct the DODAG such that the DODAG maps exactly the entire network, i.e.:

$$P(vj) \cup C(vj) \cup S(vj) = N(vj) \quad (1)$$

therefore, solving the VCP on the general graph G that maps the network is the same as solving VCP on the DODAG D, i.e.:

$$\text{VCP}(D) = \text{VCP}(G) \quad (2)$$

By definition of tree decompositions (cf. Definition 1), each vertex and edge in D is present in T, therefore

$$\text{VCP}(T) = \text{VCP}(D) \quad (3)$$

From (1), (2) and (3), we conclude that solving the VCP on the nice-tree achieves the objective of monitoring all the links of the network.

V. Conclusions & Future Work

In this paper, we investigated a strategy for increasing the robustness in connectivity of IoT for critical-time applications via a proactive monitoring of links in the DODAG constructed by RPL. We considered the Vertex Cover modeling of our problem, where the minimum number of monitoring nodes is required to cover the entire DODAG.

The VCP is a known NP-hard problem in general graphs but there exist Fixed Parameter Tractable dynamic programming algorithms for solving the VCP on tree decompositions. We developed a polynomial-time algorithm that converts the DODAG into a nice-tree decomposition with unity treewidth. This new strategy yields a significant reduction in the complexity of solving VCP on DODAGs to be only polynomial-time solvable. Meanwhile, it achieves the objective of monitoring all the links in a 6LowPAN network.

By definition of vertex cover, we now know the minimum number of required monitors to cover the entire network. However, we don't know the exact placement of the monitors. Therefore, it remains a challenge for future research to decide which nodes should be configured as monitors in order to minimize and balance the monitoring load, decrease energy consumption of the nodes and consequently maximize the lifetime of the network.

REFERENCES

[1] N. Kushalnagar, G. Montenegro, and C. Schumacher, “IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals,” 2007. [Online]. Available: https://tools.ietf.org/html/rfc4919.

[2] R. Jurdak, X. R. Wang, O. Obst, and P. Valencia, “Intelligence-Based Systems Engineering,” A. Tolk and L. C. Jain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 309–325.

[3] M. S. Kintali S, "Computing Bounded Path Decompositions in Logspace.," Electronic Colloquium on Computational Complexity (ECCC), vol. 19, p. 126, 2012.

[4] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, K. Pister, R. Struik, J. P. Vasseur, and R. Alexander, “RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks,” 2012. [Online]. Available: tools.ietf.org/html/rfc6550.

[5] H. L. Bodlaender and A. M. C. A. Koster, “Combinatorial Optimization on Graphs of Bounded Treewidth,” Comput. J., vol. 51, no. 3, pp. 255–269, 2008.

[6] O. Gaddour and A. KoubíA, “Survey RPL in a Nutshell: A Survey,” *Comput. Netw.*, vol. 56, no. 14, pp. 3163–3178, 2012.

[7] S. Raza, L. Wallgren, and T. Voigt, “SVELTE: Real-time intrusion detection in the Internet of Things,” *Ad Hoc Networks*, vol. 11, no. 8, pp. 2661–2674, 2013.

[8] H. Moser, “Exact Algorithms for Generalizations of Vertex Cover,” , M.S. thesis, Dept. Mathematics and Informatics, Friedrich-Schiller Univ., Jena, Germany, 2005.

[9] J. Alber, H. L. Bodlaender, H. Fernau, and R. Niedermeier, “Algorithm Theory - SWAT 2000: 7th Scandinavian Workshop on Algorithm Theory Bergen, Norway, July 5-7, 2000 Proceedings,” Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 97–110.

[10] N. Robertson and P. D. Seymour, “Graph minors. II. Algorithmic aspects of tree-width,” *J. Algorithms*, vol. 7, no. 3, pp. 309–322, 1986

[11] S. P. Carroll, “Domain Specific Language for Dynamic Programming on Nice-tree Decompositions,” M.S. thesis, Dept. Eng., Ohio Univ., 2013.

[12] K. D. Korte, A. Sehgal, and J. Schönwälder, “Dependable Networks and Services: 6th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2012, Luxembourg, Luxembourg, June 4-8, 2012. Proceedings,” R. Sadre, J. Novotný, P. Čeleda, M. Waldburger, and B. Stiller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 50–61.

[13] I. Dinur and S. Safra, “On the hardness of approximating minimum vertex cover,” *Ann. Math.*, vol. 162, pp. 439–485, 2005.