Wireless Netw (2015) 21:281–295
DOI 10.1007/s11276-014-0784-0

# An exact algorithm for maximum lifetime data gathering tree without aggregation in wireless sensor networks

Xiaojun Zhu · Xiaobing Wu · Guihai Chen

Published online: 9 August 2014

**Abstract** In wireless sensor networks, maximizing the lifetime of a data gathering tree without aggregation has been proved to be NP-complete. In this paper, we prove that, unless $P = NP$, no polynomial-time algorithm can approximate the problem with a factor strictly greater than 2/3. The result even holds in the special case where all sensors have the same initial energy. Existing works for the problem focus on approximation algorithms, but these algorithms only find sub-optimal spanning trees and none of them can guarantee to find an optimal tree. We propose the first non-trivial exact algorithm to find an optimal spanning tree. Due to the NP-hardness nature of the problem, this proposed algorithm runs in exponential time in the worst case, but the consumed time is much less than enumerating all spanning trees. This is done by several techniques for speeding up the search. Featured techniques include how to grow the initial spanning tree and how to divide the problem into subproblems. The algorithm can handle small networks and be used as a benchmark for evaluating approximation algorithms.

X. Zhu
College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China
e-mail: xzhu@nuaa.edu.cn

X. Zhu · X. Wu · G. Chen (✉)
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
e-mail: gchen@nju.edu.cn

X. Wu
e-mail: wuxb@nju.edu.cn

G. Chen
Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai 200240, China

**Keywords** Wireless sensor networks · Data gathering tree · Maximum lifetime · Exact algorithm · Inapproximability

## 1 Introduction

In wireless sensor networks, sensor nodes are usually battery-powered, therefore how to achieve longer lifetime is a critical issue. The energy consumption of a node is influenced by the number of messages transmitted and received, which is determined by the underlying routing structure. Among various routing structures, tree is a popular one due to low-overhead maintenance. Since there are many trees for a graph, we consider how to select the tree with maximum lifetime. We consider the scenario where sensors should forward all received data to the sink, and they cannot aggregate the received data. A node's lifetime is determined jointly by the number of descendants in the tree and its initial energy, which is different from node to node.

This problem is known to be NP-complete [14]. Some approximation algorithms are proposed, of which the best has claimed to find a lifetime within a $O(\log\log n/\log n)$ factor of the optimal, where $n$ is the number of sensor nodes [14]. Note that the problem is quite different if nodes can aggregate all received data into a single message [22]. With aggregation capability, a node's lifetime is determined by its initial energy and the number of children (not descendants), and there are good approximation algorithms and tight inapproximability results [22]. To the best of our knowledge, no inapproximability results are known for the considered problem.

Besides approximation algorithms, exact algorithms are another common approach to NP-hard optimization problems [21]. Exact algorithms for finding the maximum lifetime tree can be used in two situations. First, they can be directly applied to small practical networks to find a tree with the maximum lifetime. For example, the proposed algorithm in this paper can handle 20-node networks in several seconds. Second, they can be used to establish the optimal lifetime during the evaluations of approximation algorithms. Indeed, the performance guarantee (i.e., approximation ratio) of approximation algorithms is with respect to worst-case scenario, and it is usually helpful to understand the average-case performance. To this end, the optimal lifetime is required. The authors of [22] had to enumerate all spanning trees to establish the optimal lifetime, and, due to inefficiency in enumeration, they only considered networks with around 10 nodes. Though exact algorithms are important and necessary, none is proposed for maximum lifetime data gathering tree problem in the literature.

Our contributions are threefold. First, we prove that the problem is not only NP-hard, but also NP-hard to be approximated within a factor greater than 2/3. This result holds even if the network is homogeneous in initial energy, i.e., all sensors have equal initial energy. Homogeneous initial energy is a special case of heterogeneous initial energy, and it is intuitively easier to solve.

Second, we discover a property that can be used to decomposes the problem into subproblems. We can decompose a network graph into 2-connected subgraphs and solve the problem on each subgraph separately, followed by merging the subtrees. We prove that the resulting spanning tree is optimal. This property helps in the development of any exact algorithms.

Third, we propose an exact algorithm to solve the problem. To the best of our knowledge, this is the first nontrivial exact algorithm specifically designed for the problem. Instead of searching all spanning trees, we reduce the search space by several greedy rules. We prove that the algorithm is correct and runs in $O(mn)$ space where $m$ is the number of edges and $n$ is the number of vertices. Simulations show that our algorithm runs much faster than enumerating all spanning trees.

## 2 Related works

Much work has been devoted to choosing the routing tree structure with maximum lifetime [12, 14, 15, 18, 22]. The resulting problem is *NP*-complete in such scenarios as when sensors can aggregate all [22] or a fixed number of [12] incoming messages, or cannot perform aggregation at all [14]. When a node can aggregate all messages and the routing tree is required to be a shortest path tree, the problem is in *P* and admits very efficient exact algorithms [15, 18]. Compressive data aggregation is considered in [23].

Our problem can be cast into the *capacitated spanning tree* problem defined in [7]. Unfortunately, though the generic definition considers all kinds of graphs, most existing works in this area only focus on complete graphs such as [1, 2, 11] and their references. In sensor networks, a node can communicate with only a few nodes due to limited transmission range, so it is hard, if not impossible, to transform our problem into this restricted model. Consequently, results in this area can hardly be applied to our problem.

One exact algorithm is to enumerate all spanning trees and pick the optimal one. Then efficient enumeration of spanning trees is necessary. Read and Tarjan [17] analyze a simple backtracking-based algorithm that runs in $O(mt)$ time where $m$ is the number of edges and $t$ is the number of spanning trees. This running time is improved to be $O(nt)$ in [6] where $n$ is the number of vertices. In situations such as our problem, all nodes in a spanning tree should be scanned to compute the lifetime, so the running time of $O(nt)$ is the best achievable bound. In situations where outputting the relative changes of spanning trees is enough, the $O(t+m+n)$ time algorithm in [19] is optimal. In this work, we implement Read and Tarjan's algorithm for comparison due to its simplicity. Another general-purpose approach to solve an NP-hard optimization problem is to formulate it as an integer linear programming (ILP) problem and then use a generic ILP solver to find the optimal solution. We believe our discovery in this paper can also help give efficient ILP formulations and speed up this generic approach. Section 7 will present the comparison result of our algorithm with that of an ILP formulation.

## 3 Problem formulation

There are $n$ sensor nodes $1, 2, \ldots, n$ and an additional node 0, referred to as the sink, in the network. The network is represented by an undirected graph $G = (V, E)$ where $V$ is the set of nodes and $E$ is the set of edges. There is an edge between two nodes if and only if they can receive messages from each other. Data are routed following a tree structure rooted at the sink, and are organized into rounds. In each round, every sensor generates one message, transmits it to its parent, and relays all messages from its descendants to its parent. Sensor node $i$ has initial energy $e_i$ and we assume that the sink has unlimited initial energy, i.e., $e_0 = \infty$. Receiving a message consumes $R_x$ energy and transmitting a message consumes $T_x$ energy. We assume $T_x \geq R_x$ for practical concern. For any tree $T \subseteq G$ rooted at the sink, we denote by $d_T(i)$ the number of descendants of node $i$ in tree $T$. Then, node $i$ consumes $d_T(i)(R_x + T_x) +$

$T_x$ energy in each round. The lifetime of tree $T$ is defined as the number of rounds it can support until a node runs out of energy.

**Problem 1 (MaxLoA)** Given a network $G$ and sensor initial energy $e_1, e_2, \ldots, e_n$, find a tree $T$ to maximize the network lifetime, i.e.,

$$\max_{T \in \mathcal{S}(G)} l(T)$$

where $\mathcal{S}(G)$ is the set of spanning trees of $G$ and $l(T)$ is the lifetime of tree $T$ defined as

$$l(T) = \min_{i \geq 1} \frac{e_i}{d_T(i)(R_x + T_x) + T_x}.$$

It has been proved that MaxLoA is NP-hard, and reference [14] claimed that their algorithm provides an $\Omega(\ln \ln n / \ln n)$ approximation ratio. We will prove an inapproximability result of 2/3 and give an exact algorithm for this problem.

Most graph notations in this paper are adopted from the textbook [5]. We refer to the vertex set of a graph $G$ as $V(G)$, and its edge set as $E(G)$. For two graphs $G = (V, E)$ and $G' = (V', E')$, we set $G \cup G' = (V \cup V', E \cup E')$, and $G \cap G' = (V \cap V', E \cap E')$. For a subset $F \subseteq E$, we write $G - F$ for $(V, E \setminus F)$, and $G + F$ for $(V, E \cup F)$. If $v$ is a vertex of $G$, we write $G - v$ for the graph obtained from $G$ by deleting vertex $v$ and all its incident edges.

Some notations are less formal, and their usage is restricted to this paper. For a graph $G$, we write $G + (a, b)$ for $G \cup (\{a, b\}, \{(a, b)\})$, i.e., $G + (a, b)$ is the graph obtained by inserting edge $(a, b)$ and (possibly) its ends to $G$. A *partial spanning tree* $P$ of $G$ is a connected subgraph of $G$ that is a tree. This is different from other definitions (e.g., [17]) where a partial spanning tree is simply considered as a set of edges, and can be disconnected. One consequence is that, if we insert an edge to a partial spanning tree, then we also add exactly one vertex to it at the same time. As mentioned before, by $\mathcal{S}(G)$ we denote the set of spanning trees of $G$. For a graph $G$ and a partial spanning tree $P \subseteq G$, we let

$$\mathcal{S}_P(G)$$

be the set of spanning trees of $G$ containing $P$ as a subtree. We note that $\mathcal{S}_{P'}(G) \subseteq \mathcal{S}_P(G)$ if $P' \supseteq P$, and $\mathcal{S}_P(G') \subseteq \mathcal{S}_P(G)$ if $G' \subseteq G$. Thus, to reduce the set $\mathcal{S}_P(G)$, we can either grow $P$ or reduce $G$.

We emphasize that, in this paper, all spanning trees and partial spanning trees are implicitly *rooted* trees, because different roots lead to different lifetimes and it is necessary to specify the root. In our algorithm, we may encounter new subproblems whose underlying graphs do not contain the sink node. In this case, we are not able to compute the lifetime of any spanning tree for the subproblem if the corresponding root node is not specified. To this end, whenever we encounter a new subproblem, we will *assign* it a sink node, and then the spanning trees and partial spanning trees for the subproblem are (implicitly) rooted at the assigned sink node. Though an assigned sink node may have limited initial energy, making it different from the network sink, we will not distinguish between them unless it is necessary.

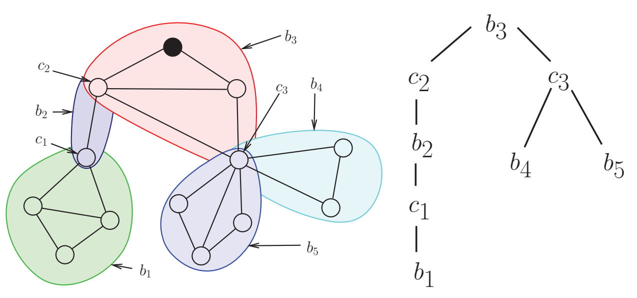

**Fig. 1** The block structure of a graph (*left*) and the rooted block tree (*right*). The *solid circle* represents the sink. The three cutvertices $c_1, c_2, c_3$ divide the graph into five blocks $b_1, \ldots, b_5$. Nodes in block $b_1$ except $c_1$ are descendants of $c_1$ in all spanning trees, so $c_1$ is the sink of $b_1$. Similarly, $c_2$ is the sink of $b_2$, and $c_3$ is the sink of both $b_4$ and $b_5$. The sink (*solid circle*) of the network is the sink of $b_3$

## 4 Reducing the search space

One approach to find an optimal spanning tree is to scan all spanning trees, but the search space may be very large. In this section, we present techniques for clever exploration of the search space. We first show how to decompose the problem into subproblems, and then give some rules to exclude search space that is unnecessary for checking.

### 4.1 Decomposing the problem into subproblems

Consider the following example. A graph $G$ consists of two disjoint subgraphs that are connected by a bridge. Clearly, the bridge should be included into any spanning tree. Let the number of spanning trees of two subgraphs be $x$ and $y$ respectively. Then the total number of spanning trees of $G$ is $x \cdot y$. Suppose we find for each subgraph an optimal spanning tree, then the two found trees, together with the bridge, form an optimal spanning tree for $G$. But in this case, we need to scan only $x + y$ trees. This example illustrates the basic idea of decomposition.

Generally, any cutvertex can help divide the graph into subgraphs with each leading a subproblem. Solving these subproblems gives us a desired optimal spanning tree. When there are no cutvertices, we will design rules to reduce the size of graph so that new cutvertices will appear. In a graph containing cutvertices, we can decompose the graph into *blocks* defined below and find optimal spanning trees for each block, followed by merging these found spanning trees.

**Definition 1** (*block* [5]) A block of a graph is a maximal connected subgraph without a cutvertex. It is either a maximal 2-connected subgraph, or a bridge (with its ends), or an isolated vertex.

An example is given in the left of Fig. 1, where blocks are indicated by shaded regions. Different blocks overlap in at most one vertex, which is a cutvertex.

**Definition 2** *(block graph)* For a graph $G$, let $C$ be the set of cutvertices and $B$ be the set of blocks. A *block graph* is a bipartite graph on $C \cup B$ such that there is an edge $(c, b)$ joining $c \in C$ and $b \in B$ if and only if $c$ is a vertex of block $b$.

**Lemma 1** ([5]) *The block graph of a connected graph is a tree.*

Thus, we also refer to block graph as block tree, and root the block tree at either the sink or the block containing the sink whichever applies. Then in this rooted tree, the parent of each block is a cutvertex, and is referred to as *parent cutvertex*. A block will correspond to a subproblem, and as mentioned before, new subproblems may not contain the sink node, so *we assign a block's parent cutvertex as its sink node*. This assignment is fine because the traffic from nodes in a block can only be forwarded by the block's parent cutvertex. The right side of Fig. 1 gives the rooted block tree of the example on the left. We can decompose any spanning tree of a graph as the union of spanning trees of its blocks.

**Lemma 2** *Suppose a graph $G$ can be decomposed into blocks $b_1, b_2, \ldots, b_t$ with sinks $s_1, s_2, \ldots, s_t$. Suppose the rooted bock tree is on $C \cup B$, where $C$ is the set of cutvertices and $B$ is the set of blocks. Let $P$ be a partial spanning tree of G, and $T \in \mathcal{S}_P(G)$ be a spanning tree. For $i = 1, \ldots, t$, let $T_i = T \cap b_i$, and $P_i = P \cap b_i$. Then we have the following three results.*

1. $T = \cup_i T_i$;
2. *$T_i$ is a spanning tree of $b_i$, and contains $P_i$ as a subtree;*
3. *In all spanning trees in $\mathcal{S}_P(G)$, cutvertex $c \in C$ is an ancestor of all nodes belonging to some block that is a descendant of $c$ in the rooted block tree.*

*Proof* To see (1), simply note that the union of all $b_i$ is exactly $G$. To prove (2), we show that all simple paths in $T$ connecting vertices in $b_i$ are in $T_i$ so that $T_i$ is a connected subgraph spanning $b_i$. In fact, all simple paths in $G$ connecting two vertices in $b_i$ can only consist of edges in $b_i$. Otherwise, there exists a simple path connecting two vertices in $b_i$ and this path intersects with $b_i$ at only these two vertices (such a path is named $b_i$-path in [5]). In this case, the union of the path and $b_i$ is a larger block, contradicting the maximality of $b_i$. For (3), removing $c$ disconnects the sink with block $b_j$ if $b_j$ is a descendant of $c$ in the rooted block tree. Therefore, nodes in $b_j$ can only transmit data through $c$ so that they are descendants of $c$ in any spanning tree. □

To make each block a subproblem, we need to redefine the lifetime of spanning trees of a block by considering nodes in other blocks. According to Lemma 2, the descendants of a cutvertex $c \in T_i$ include those in $T_i$ and all nodes in $\cup_{b_j \in DES(c)} b_j$, where $DES(c)$ is the set of blocks that are descendants of $c$ in the rooted block tree. All ancestors of $c$ in $T_i$ are also ancestor of these nodes. For the example in Fig. 1, the number of descendants of cutvertex $c_3$ is the number of its descendants in $b_3$ plus 5 (two of them are from $b_4$ and the rest three are from $b_5$). Using this modified lifetime computation, we get the following theorem.

**Theorem 1** *Given a graph G and a partial spanning tree P, suppose the graph can be decomposed into blocks $b_1, b_2, \ldots, b_t$ with sinks $s_1, s_2, \ldots, s_t$. For $i = 1, \ldots, t$, let $T_i^* \in \mathcal{S}_{P_i}(b_i)$ be the tree with the maximum lifetime among trees in $\mathcal{S}_{P_i}(b_i)$. Then $T^* = \cup_{i=1}^t T_i^*$ is a tree in $\mathcal{S}_P(G)$ with the maximum lifetime.*

*Proof* Suppose $T^*$ is not optimal. Then there exists a tree with greater lifetime. According to Lemma 2, we can decompose this tree into spanning trees for the blocks. One of these spanning trees, say $T_i$, must have a lifetime greater than that of $T_i^*$, contradicting the optimality of $T_i^*$. □

In the graph theory related literature, decomposing graphs into blocks may be a known divide-and-conquer technique, but Lemma 2 and Theorem 1 generalize this known technique by allowing different partial spanning tree $P$, and, to the best of our knowledge, the generic literature has considered at most the case when $P = \emptyset$. Allowing different partial spanning trees helps the design of our exact algorithm, because graph decomposition alone cannot directly give us an efficient algorithm and we need to incorporate other procedures. This incorporation is done via the partial spanning tree $P$.

Therefore, we can decompose the graph into blocks, and find an optimal spanning tree for each block, followed by merging them to produce the desired spanning tree. The benefit is evident in terms of the number of scanned trees.

**Fact 1** $|\mathcal{S}_P(G)| = \prod_{i=1}^{t} |\mathcal{S}_{P_i}(b_i)|$ and $\left|\cup_{i=1}^{t}\mathcal{S}_{P_i}(b_i)\right| = \sum_{i=1}^{t} |\mathcal{S}_{P_i}(b_i)|$.

Because a spanning tree of a block is smaller than the spanning tree of the graph, one can check that scanning $\cup_{i=1}^{t}\mathcal{S}_{P_i}(b_i)$ never takes longer time than scanning $\mathcal{S}_P(G)$, even in cases where $G$ is itself a tree such that $|\mathcal{S}_P(G)| = 1$ and $|\cup_{i=1}^{t} \mathcal{S}_{P_i}(b_i)| = |V(G)| - 1$. The fundamental reason is that decomposing a graph into blocks can be done in linear time by a depth-first search from the sink, which is proposed in [9] and now a textbook exercise [4].

## 4.2 Scanning less spanning trees

The blocks obtained by Sect. 4.1 cannot be further decomposed, so graph decomposition is useful (in reducing the search space) for at most once for a given graph and a given partial spanning tree. This subsection considers the situation when graph decomposition has been applied and we have not found the optimal tree. We will introduce several procedures that either grow the partial spanning tree or reduce the graph in a smart way.

The following fact is the last resort we have towards an exact algorithm. For a partial spanning tree $P$, consider an edge from $V(P)$ to $V(G) \setminus V(P)$. A spanning tree either contains this edge, or does not contain. So the set $\mathcal{S}_P(G)$ can be divided into two disjoint parts.

**Fact 2** For a graph $G$ with a partial spanning tree $P$, suppose we have an edge $(a, b)$ with $a \in V(P)$ and $b \notin V(P)$. Then $\mathcal{S}_P(G) = \mathcal{S}_{P'}(G) \cup \mathcal{S}_P(G')$, where $P' = P + (a, b)$, and $G' = G - (a, b)$.

All trees in $\mathcal{S}_P(G)$ containing edge $(a, b)$ belong to $\mathcal{S}_{P'}(G)$, and the rest trees are in $\mathcal{S}_P(G')$. Note that when $(a, b)$ is a bridge of $G$, the set $\mathcal{S}_P(G')$ is empty since $G'$ becomes disconnected, but this situation will never occur if $G$ is a non-bridge block obtained by graph decomposition in Sect. 4.1. Observe that $P'$ is larger than $P$ and $G'$ is smaller than $G$. Thus applying Fact 2 either grows $P$ or reduces $G$. The problem with Fact 2 is that each checked edge results in two subproblems and repeatedly applying Fact 2 explores the whole space of $\mathcal{S}_P(G)$, which is not efficient. Consequently, Fact 2 will only be used as a last resort. We give some procedures that include or exclude several edges directly without generating two subproblems each time, speeding up the enumeration process.

First, we give a procedure that grows any partial spanning tree $P$. We emphasize that $P$ could be *any* partial spanning tree. This procedure will be applied before other procedures.

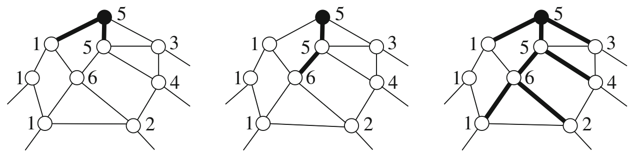

**Fig. 2** Illustration of procedure grow. The *solid circle* represents the sink, and the number around a node is the initial energy. The leftmost is the input graph with *bold edges* indicating the given partial spanning tree $P$. In the middle figure, bold edges are elements of $X$ after step 3 of procedure grow. *Bold edges* in the rightmost figure indicate the final $X$

*Procedure Grow* Let $s$ be the sink. Given any partial spanning tree $P$, let $X$ be the partial spanning tree obtained by the following steps:

1. Include sink $s$ into $X$;
2. For an edge $(a, b)$ of $P$ such that $a \in V(X), b \notin V(X)$ and $e_b \geq e_s$, set $X = X + (a, b)$. Repeat (2) until no edge can be added;
3. For an edge $(a, b)$ of $G$ such that $a \in V(X), b \notin V(X)$ and $e_b \geq e_s$, set $X = X + (a, b)$. Repeat (3) until no edge can be added;
4. For all vertex $a$ of the tree $X$ obtained after step (3), insert all edges $(a, b)$ to $X$ where $b \notin V(P) \cup V(X)$;
5. Let $X$ be $X \cup P$.

Note that step 2 should not be omitted. Otherwise, at step 5 $X \cup P$ may contain cycles, and if we in turn omit step 5 to avoid cycle, then it is possible that $X$ may contain less edges than $P$, contradicting our purpose of "growing" $P$. Figure 2 gives an example with the leftmost indicating $P$ and the rightmost indicating the final $X$. Note that the sink here has finite energy 5. In this example, five more edges are directly added to the partial spanning tree.

It is easy to see that $X$ is a tree, i.e., it does not contain cycles, because each node in $X$ is included only once. In the case where the sink has infinity energy, this procedure only includes edges incident with the sink into $X$, but in subproblems such as Fig. 2 where the sink has finite energy, this procedure includes more edges. We prove that this procedure is safe in that the maximum lifetime of excluded spanning trees is no larger than the maximum lifetime of remaining spanning trees.

**Lemma 3** *Given partial spanning tree $P$, suppose $X$ is obtained by Procedure Grow, then the maximum lifetime of trees in $\mathcal{S}_X(G)$ is equal to the maximum lifetime of trees in $\mathcal{S}_P(G)$.*

*Proof* Since $\mathcal{S}_X(G) \subseteq \mathcal{S}_P(G)$, we will show that each tree in $\mathcal{S}_P(G)$ can be modified to be a tree in $\mathcal{S}_X(G)$ without reducing its lifetime. Thus the lemma follows easily. Note that all nodes included into $X$ prior to Step 3 have initial

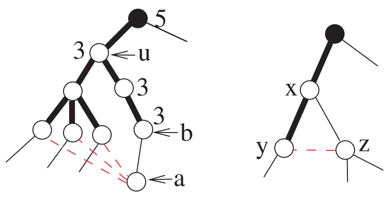

**Fig. 3** Examples of Procedure Reduce-1 (*left*) and Reduce-2 (*right*). *Solid circle* represents the sink, and the number around a node is the initial energy. *Bold edges* are edges in the partial spanning tree. Note that the poorest ancestor of $b$ is node $u$ rather than the node between $u$ and $b$. *Dashed edges* in the two figures are deleted in Reduce-1 and Reduce-2 respectively

energy greater than or equal to the sink. Therefore, no matter how many descendants they have, their lifetime is greater than that of the sink, and the sink's lifetime is an upper bound of spanning tree's lifetime. Therefore, for any tree $T$ in $\mathcal{S}_P(G) \setminus \mathcal{S}_X(G)$, if a node in $X$ has a parent in $T$ different from the parent in $X$, we can change the parent to be the same as in $X$. Repeating this process produces a tree in $\mathcal{S}_X(G)$ and the lifetime is not reduced. This completes the proof. □

Second, we consider how to reduce the size of a graph directly. Given a partial spanning tree $P$, we classify nodes in $P$ into two categories, *rich* and *poor*. A node is *poor* if its initial energy is strictly less than all its ancestors' initial energy. Otherwise, the node is *rich*. Rich nodes will not have the smallest lifetime among nodes, since its lifetime is greater than that of some ancestor. We refer to $u$ as the *poorest ancestor* of node $v$ if $u$ is the one with the least initial energy among ancestors of $v$. If several ancestors all have the least energy, then we refer to the one that is closest to the sink.

*Procedure Reduce-1* Given a graph $G$, a partial spanning tree $P$, a rich node $b \in V(P)$ with its poorest ancestor $u$, if there exists an edge $(a,b) \in G$ with $a \notin V(P)$, then we can delete from $G$ all edges in $\{(a,c) \mid c \,\text{is a descendant of u inP}\}$ except $(a,b)$. Let $G'$ be the resulting graph.

*Procedure Reduce-2* Suppose $(x,y)$ is in the partial spanning tree $P$ with $x$ as the parent of $y$, then for any node $z \notin P$ adjacent to both $x$ and $y$, let $G'$ be the graph obtained by removing edge $(y,z)$ from $G$.

Figure 3 gives examples of Reduce-1 and Reduce-2. Dashed edges are deleted from the graph.

**Lemma 4** *For the graph $G'$ obtained by Procedure Reduce-1 or Reduce-2, the maximum lifetime of trees in $\mathcal{S}_P(G')$ is equal to the maximum lifetime of trees in $\mathcal{S}_P(G)$.*

*Proof* Since $\mathcal{S}_P(G') \subseteq \mathcal{S}_P(G)$, we will show that each tree in $\mathcal{S}_P(G)$ can be modified to be a tree in $\mathcal{S}_P(G')$ without reducing lifetime. Consider the case of Reduce-1. For any tree $T \in \mathcal{S}_P(G)$, if the parent of node $a$ is not a descendant of $u$ in $P$, or is exactly $b$, then $T \in \mathcal{S}_P(G')$. Otherwise, if the parent of $a$ in $T$ is a descendant of $u$ in $P$, then we can change $a$'s parent to be $b$. In this case, the lifetime of $u$ is unchanged, and the lifetime of $b$, as well as the nodes between $u$ and $b$, is still greater than $u$'s lifetime. Consequently, the lifetime of the new tree is not reduced, but the new tree is now in $\mathcal{S}_P(G')$. For the case of Reduce-2, for any tree $T$ in $\mathcal{S}_P(G) \setminus \mathcal{S}_P(G')$, we have $(y,z) \in T$. Remove $(y,z)$ from $T$ and add the edge $(x,z)$. We get a tree in $\mathcal{S}_P(G')$ without reducing the lifetime. This completes the proof. □

*Procedure Reduce-3* For a graph $G$ and a partial spanning tree $P$, let $R(P)$ denote the set of redundant edges (edges not in $P$ joining vertices of $P$). Let $G' = G - R(P)$.

**Fact 3** Let $G'$ be the graph from Procedure Reduce-3, then $\mathcal{S}_P(G) = \mathcal{S}_P(G')$.

Intuitively, edges in $R(T)$ will not appear in any spanning tree, otherwise a cycle occurs. Repeatedly applying Lemmas 3 and 4 and Fact 3 gives the following theorem.

**Theorem 2** *For graph G and partial spanning tree P, suppose after any sequence of Procedure Grow, Reduce-1, Reduce-2, Reduce-3, the new graph is $G'$ and the new partial spanning tree is $P'$, then the maximum lifetime of trees in $\mathcal{S}_{P'}(G')$ is equal to the maximum lifetime of trees in $\mathcal{S}_P(G)$.*

It is worth noting that both Theorem 1 and Theorem 2 hold if all sensors have heterogeneous traffic demand in each round (instead of sending one message in a round). Therefore, the exact algorithm in the next section also applies to this general situation.

## 5 The proposed exact algorithm for finding an optimal spanning tree

In this section, we present our algorithm that guarantees to find an optimal spanning tree. The basic idea is to use the properties discovered in the previous section whenever possible. For example, we perform decomposition when the current graph may not be 2-connected resulted from deleted edges. If a new subproblem is identified by the decomposition, then Procedure Grow is performed to quickly build a partial spanning tree. If an edge is included, Procedures Reduce-1, Reduce-2 and Reduce-3 are performed to reduce the size of the graph. The algorithm is described in Algorithms 1, 2 and 3.

In Algorithm 1, we initialize the partial spanning tree by including the sink and its incident edges. This is a degenerated case of Procedure Grow, since the sink has infinite energy. Then we perform Reduce-3. (No edge will be removed by performing Reduce-1 or Reduce-2.) At last, we decompose the problem by calling Algorithm 2.

Algorithm 2 decomposes a graph into blocks, solves each block and merges the solutions. It is designed according to Theorem 1. For a block consisting of one edge, this edge is its optimal spanning tree. Otherwise, the block is a 2-connected subgraph. Two cases are considered. If its sink is the same as the input graph, then we have already performed Procedure Grow for this sink so that we call Algorithm 3. Otherwise, we perform Procedure Grow and then either recursively call Algorithm 2 if some edges are deleted, or call Algorithm 3. The reason for this design is that, if some edges are deleted, then it is possible that we can further decompose the graph. But if no edge is deleted, then the subgraph is still 2-connected and further decomposition is impossible.

Algorithm 3 implements Fact 2, which tries including an edge to the tree. Note that the input graph to Algorithm 3 is always 2-connected, so we will always find such an edge in Line 1. In the first case (Lines 2-14), we include the edge into the partial spanning tree. Procedures Reduce-1, Reduce-2, and Reduce-3 are performed to quickly reduce the size of the graph. Similarly, if some edge is deleted, then it is possible that we may decompose the graph so that we call Algorithm 2; if no edge is deleted, then the graph is still 2-connected and it is impossible to further decompose the graph, for which we recursively call Algorithm 3. In the second case (Lines 17-18), we delete the edge from the graph, and call Algorithm 2.

**Algorithm 1:** ExactMaxLoA

```
1 include the sink, its incident edges, and the nodes at the other ends of these edges into the partial spanning tree; // Procedure Grow
2 delete $R(P)$ from $G$; // Procedure Reduce-3
3 $(T, l) \longleftarrow decomp(G, s)$;
4 $T$ is the desired tree;
```

**Algorithm 2:** Function *decomp*. Decompose the graph into blocks and conquer each.

```
   Input: graph $G$, sink $s$
   Output: an optimal spanning tree $T$ and its lifetime $l$
1  decompose graph $G$ with sink $s$ into blocks $b_1, b_2, \ldots, b_t$ with sinks $s_1, s_2, \ldots, s_t$ using the algorithm in [10];
2  for $i \longleftarrow 1$ to $t$ do
3      if $b_i$ contains only one edge then // a bridge
4          $T_i = b_i$;
5          $l_i \longleftarrow \frac{e_v}{d \cdot R_x + (d+1)T_x}$ where $v$ is the other node in $b_i$ rather than $s_i$, and $d$ is the number of descendants of $v$;
           /* d is maintained during decomposing a graph into blocks */
6      else if $s_i = s$ then /* already performed Procedure Grow */
7          $[T_i, l_i] = try(b_i, s_i)$;
8      else /* possible to grow the tree by Procedure Grow */
9          construct $X$ using Procedure Grow for graph $b_i$ with sink $s_i$ and partial spanning tree $P_i$;
10         $B_1 \longleftarrow E(X) \setminus E(P_i)$;
11         add $B_1$ to $P_i$;
12         $B_2 \longleftarrow R(P)$; // redundant edges
13         delete $B_2$ from $b_i$; // Procedure Reduce-3
14         if $B_2 \neq \emptyset$ then /* some edges are deleted, further decomposition is possible */
15             $(T_i, l_i) \longleftarrow decomp(b_i, s_i)$;
16         else
17             $(T, l) \longleftarrow try(b_i, s_i)$;
           /* The following restores $b_i$ */
18         add $B_2$ to $b_i$;
19         delete $B_1$ from $P_i$;
20 $l \longleftarrow \min_{i=1}^{t} l_i$;
21 $T \longleftarrow \bigcup_{i=1}^{t} T_i$;
22 return $T, l$;
```

**Algorithm 3:** Function *try*. Try including an edge.

```
Input: graph $G$, sink $s$
Output: an optimal spanning tree $T$ and its lifetime $l$
perform BFS from the sink, let $(x, y)$ be the first encountered edge not in $P$ with $x \in P$;
/* the first case, include edge $(x, y)$ */
include edge $(x, y)$ and vertex $y$ into $P$;
$B_1 \longleftarrow R(P)$; // redundant edges
delete $B_1$ from $G$; // Procedure Reduce-3
$B_2 \longleftarrow \{(y, z) \in G \setminus P \mid z \notin P, (x, z) \in G \setminus P\}$;
delete $B_2$ from $G$; // Procedure Reduce-2
if $y$ has initial energy greater than or equal to any ancestor then // is rich
    let $y$'s poorest ancestor be $u$;
    let $B_3$ be $\{(a, c) \mid a \notin P, c \neq y, c$ is a descendant of $u\}$ where $a$ is adjacent to $y$;
    delete $B_3$ from $G$; // Procedure Reduce-1
if $B_1 \bigcup B_2 \bigcup B_3 \neq \emptyset$ then
    /* have deleted some edges, $G$ may not be 2-connected */
    $[T_1, l_1] \longleftarrow decomp(G, s)$;
else
    /* no edge has been deleted, $G$ is still 2-connected */
    $[T_1, l_1] \longleftarrow try(G, s)$;
/* restore the graph */
add edges in $B_1 \bigcup B_2 \bigcup B_3$ to $G$;
delete $(x, y)$ and $y$ from $P$;
/* the second case, exclude $(x, y)$ */
delete $(x, y)$ from $G$;
$[T_2, l_2] \longleftarrow decomp(G, s)$;
add $(x, y)$ to $G$;
let $l$ be the greater of $l_1$ and $l_2$ and $T$ be the corresponding tree;
return $T, l$;
```

For the data structure, nodes and edges are global variables that can be modified by any algorithm (this is why we need to restore the graph every time we make modifications). A node has a flag field indicating whether it is in the partial spanning tree, a link to its parent if it is in the partial spanning tree, and a doubly-linked list of adjacent edges. Each edge also has a flag field indicating the membership of the partial spanning tree. Edges are stored by doubly-linked lists because in such data structure, inserting or deleting an edge can be done in constant time. Since our graph is undirected, there are two directed edges representing an undirected edge. Each directed edge has a *pair* field linked to the directed edge on the reverse direction. Whenever we insert or delete an undirected edge, we insert or delete both directed edges.

## 5.1 Theoretical analysis

We first show the correctness of the algorithm, and then analyze the space requirement.

**Lemma 5** *Algorithm 1 finds a spanning tree of the maximum lifetime in finite time.*

*Proof* We claim that *decomp* finds an optimal spanning tree in the set $\mathcal{S}_P(G')$ in finite time where $P$ and $G'$ are the input to *decomp*. Since Algorithm 1 calls *decomp* after two operations allowed by Theorem 2, which suggests that the maximum lifetime of trees in $\mathcal{S}_P(G')$ is equal to that of trees in $\mathcal{S}(G)$. The theorem follows immediately.

To prove the claim, we will use induction twice. The first is on the number $n$ of vertices in $G'$, and the second is an embedded induction on the number $l$ of edges in $G'$ not in the partial spanning tree.

Induction on $n$. (Base Case) When $n = 2$, the graph can only be a bridge. The decomposition at Line 1 of *decomp* yields one block. No matter what $l$ is, *decomp* solves the problem at the **if** block at Line 3. (Induction Hypothesis-1) Now assume the claim holds for all $n < h$. Consider the case when $n = h$.

Induction on $l$ (for $n = h$). (Base Case) When $l = 0$, all edges in the graph are included in the partial spanning tree. The graph can only be a tree, because whenever a new edge is included into the partial spanning tree, Procedure Reduce-3 is performed so that no redundant edges are left (e.g., Line 3 of *try*). Consequently, the blocks at Line 1 of *decomp* are bridges and the algorithm will compute the lifetime at the **if** block at Line 3. The claim holds. (Induction Hypothesis-2) Suppose the claim holds for all $l < k$ when $n = h$. Then consider the case when $l = k$.

(1) If there is only one block at Line 1 of *decomp*, then the input graph is 2-connected, and Line 7 is executed to find the solution. We show that *try* will solve the problem by calling *decomp* with smaller $l$. In *try*, the search space is divided into two parts

based on Fact 2. If we can find an optimal tree for each part, we can find the desired tree. In the first part (Lines 1–16), note that it is safe to change Lines 11–14 to a single call as "$[T_1, l_1] \longleftarrow decomp(G, s)$". The reason is that, if we call *decomp* at Line 14, it will still call *try* without any modification to the graph and the partial spanning tree (it just performs an unnecessary decomposition). Since one edge is added into the partial spanning tree, we have $l < k$ so that *decomp* will find the desired tree at the first search space by Induction Hypothesis-2. Note that the operations of Lines 2–10 do not exclude all optimal spanning trees in the search space due to Theorem 2. In the second part (Lines 17–19), the call of *decomp* will return the desired tree by Induction Hypothesis-2, since one edge is deleted from the graph so that $l < k$.

(2) If there are several blocks at Line 1 of *decomp*, then each block has vertices less than $h$. We show that we will find for each block an optimal tree so that the merged tree at Line 21 of *decomp* is optimal due to Theorem 1. It is trivial if a block is a bridge. If a block has the same sink as the original one, then *try* is called at Line 7. Based on similar analysis as (1) by using Induction Hypothesis-1, we can solve this case in finite time. The left case is when the sink is different from the original one. Note that we can also change *try* to *decomp* at Line 17. This only changes the running time and does not influence the result, since if we call *decomp* at Line 17, it will also call *try* without modifying the graph and the partial spanning tree. After changing, we call *decomp* with less vertices than $h$. By Induction Hypothesis-1, we can find an optimal tree in the search space in finite time.

Therefore, Induction Hypothesis-2 also holds for $l = k$ when $n = h$, which completes the nested induction. In turn, Induction Hypothesis-1 holds for $n = h$. This completes the proof. □

For the space requirement, we have the following lemma.

**Lemma 6** *Algorithm 1 can be implemented to run in O(mn) space where m is the number of edges and n is the number of nodes in the network.*

*Proof* Consider any recursion path. It is a sequence of *decomp* and *try*. We first show that the depth of recursion, i.e., the length of the recursion path, is at most 2 m. We claim that every two consecutive calls either reduce the number of edges in the graph by at least 1, or include at least one edge to the partial spanning tree. There are four types of such consecutive calls, $decomp \rightarrow decomp, decomp \rightarrow try, try \rightarrow try$, and $try \rightarrow decomp$. Since *try* will either grow the partial spanning tree by 1 or remove an edge, we only need to consider $decomp \rightarrow decomp$. According to Algorithm 2, such situation only happens when we have deleted some edges ($B_2 \neq \emptyset$). Therefore, the claim holds. Since the graph contains $m$ edges, and each edge is either deleted or included into the final tree by at most two calls, the total number of calls in the recursion path is at most 2 m.

Next, consider the sets $B_2$ in *decomp*, and $B_1, B_2, B_3$ in *try*. Each deleted edge only belongs to at most one of these sets in the recursion path, since deleted edges will not appear in subsequent calls. For the set $B_1$ in *decomp*, any node or edge belongs to at most one such $B_1$ in the recursion path. The space requirement is in $O(m)$.

The $O(mn)$ space is due to the decomposition. The content of the sets $b_i$ in *decomp* consists of deleted edges so that no extra storage is consumed. However, the variables themselves may consume storage. The number of variables $b_i$ is bounded by $O(n)$. The partial solutions $T_i$ of *decomp*, and the solution $T_1$ of *try* are also bounded by $O(n)$ in size. The space requirement is in $O(mn)$ due to the $O(m)$ recursion depth. This completes the proof. □

Combining Lemmas 5 and 6 gives the following theorem.

**Theorem 3** *Algorithm 1 finds an optimal tree in finite time, and it requires $O(mn)$ space where m is the number of edges and n is the number of nodes in the network.*

Note that our algorithm runs in exponential time in the worst case. This is reasonable since no polynomial-time algorithm can solve the problem unless $P = NP$.

## 6 Hardness results

In this section, we prove that it is NP-hard to approximate the problem with a factor greater than $2/3$. We first prove that the problem is inapproximable to a factor greater than $5/7$ unless $P = NP$, and then refine the proof to get the inapproximability result of $2/3$. Note that directly merging the two proofs gives the inapproximability result of $2/3$, but the resulting proof is lengthy and we feel that splitting it into two parts helps understanding. In the proofs, we reduce from a well-known NP-hard problem, three dimensional matching (3DM) defined as follows.

*Problem 2* *(3DM* [7]*)* Given a set $M \subseteq A \times B \times C$ where $A$, $B$ and $C$ are disjoint sets having the same number $q$ of elements, does $M$ contain a matching $M' \subseteq M$ such that $|M'| = q$ and no two elements of $M'$ agree in any coordinate?

We only consider the instances of 3DM where $|M| > q$ and each element in $A \cup B \cup C$ appears at least once in a triple in $M$. It is easy to see that the resulting problem is still NP-hard, because all the excluded instances can be trivially solved. This restriction of 3DM ensures that the networks of the constructed MaxLoA instances in the following proofs are connected.

**Theorem 4** *MaxLoA cannot be approximated by any polynomial algorithm to a factor >5/7, unless* $P = NP$.

*Proof* We show that any such approximation algorithm for MaxLoA can be used to solve 3DM. Our reduction is inspired by [13], and a similar (and weaker) reduction is used to show the hardness of finding maximum lifetime tree when the trees are restricted to be shortest path trees [18].

Given an instance of 3DM, construct the following instance of MaxLoA. The network consists of $3|M| + 1$ nodes divided into four layers according to the hop distance to the sink. See Fig. 4 for an example. The sink alone is at layer 0. At layer 1, there are $|M|$ tuple nodes $m_1, m_2, \ldots, m_{|M|}$ corresponding to the tuples in $M$ and connected to the sink. At layer 2, there are $2q$ element nodes corresponding to the $2q$ elements of $B \cup C$ and $|M| - q$ additional dummy nodes. An element node is connected to a tuple node if and only if the element is in the tuple. The $|M| - q$ dummy nodes are constructed as follows. For each $a \in A$, suppose the number of tuples in $M$ containing $a$ is $t_a$. Then we construct $t_a - 1$ dummy nodes and connect each of them to all these $t_a$ tuples. Every dummy node at layer 2 has a distinct dummy node neighbor at layer 3. Since each element in $B \cup C$ appears at least once in $M$ (due to our restriction of 3DM mentioned before), the network is connected. Let $T_x = R_x = 1$ and the initial energy of every node to be 5. Our construction is now complete, and it can certainly be done in polynomial time. We will prove that the 3DM has a matching if and only if the constructed MaxLoA has a spanning tree with a lifetime of 1.

On one hand, suppose the 3DM has a matching $M'$. Then we can find a tree $T$ in the constructed MaxLoA instance as follows. Include to $T$ all edges between nodes in layer 0 and nodes in layer 1, and edges between nodes in layer 2 and nodes in layer 3. For each node $m_l$ at layer 1 with $m_l = (a, b, c) \in M'$, include to $T$ the two edges $(m_l, b)$ and $(m_l, c)$. Now for any $a \in A$, among the $t_a$ tuples in $M$ containing $a$, only one has two children. We find an arbitrary perfect matching from the rest $t_a - 1$ tuples to the $t_a - 1$ dummy nodes constructed for $a$, and include the matching to tree $T$. It is easy to see that $T$ is a spanning tree, and each node at layer 1 has exactly two descendants, so it has a lifetime of 1.

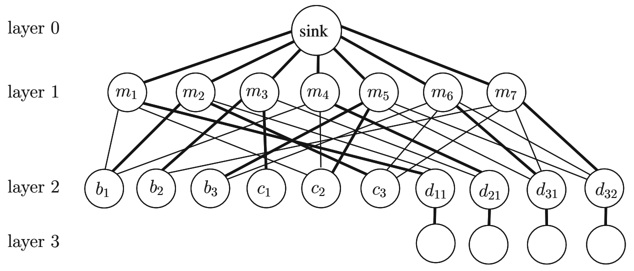

**Fig. 4** A constructed MaxLoA instance for the following 3DM instance: $A = \{a_1, a_2, a_3\}$, $B = \{b_1, b_2, b_3\}$, $C = \{c_1, c_2, c_3\}$ and $M = \{m_1, m_2, m_3, m_4, m_5, m_6, m_7\}$ with $m_1 = (a_1, b_1, c_2)$, $m_2 = (a_1, b_1, c_3)$, $m_3 = (a_2, b_2, c_1)$, $m_4 = (a_2, b_1, c_2)$, $m_5 = (a_3, b_3, c_2)$, $m_6 = (a_3, b_3, c_3)$, and $m_7 = (a_3, b_2, c_3)$. Node $d_{ij}$ is the $j-$th dummy node corresponding to $a_i$. The 3DM instance admits a matching $M' = \{m_2, m_3, m_5\}$. The corresponding spanning tree for the MaxLoA instance is indicated by thick edges. We can see that each node at layer 1 has exactly two descendants so that the tree has a lifetime of 1

On the other hand, suppose there exists a spanning tree $T$ for the constructed MaxLoA with lifetime of 1. Then we can find a matching $M'$ as follows. Observe that in $T$, every tuple node can have at most two descendants. Consequently, exactly $|M| - q$ tuple nodes have a dummy node at layer 2 as child, since each dummy node at layer 2 contributes two descendants. The rest $q$ tuple nodes are the desired matching $M'$. To see why $M'$ is a matching, note that the rest $q$ tuple nodes need to serve the $2q$ element nodes of $B \cup C$ so that any two tuples cannot agree in either coordinate $B$ or $C$. Consider coordinate $A$. Suppose two tuples in $M'$ both contain element $a$ for some $a \in A$. Since both of them do not have dummy nodes as children and there are totally $t_a$ tuples containing $a$, the rest $t_a - 2$ tuple nodes have to be parent of the $t_a - 1$ dummy nodes at layer 2 constructed for $a$, contradicting the observation that all tuple nodes in $T$ have at most one dummy node at layer 2 as child.

At last, we prove that, for a spanning tree $T$ for the constructed MaxLoA, $l(T) = 1$ iff (if and only if) $l(T) > 5/7$, which means that the mentioned approximation algorithm can tell whether $l(T) \geq 1$. To see this, suppose $l(T) > 5/7$. Solving the inequality $5/(d_T(i) + d_T(i) + 1) > 5/7$ yields $d_T(i) \leq 2$. Thus each node at layer 1 has at most two descendants, and the tree has a lifetime of 1. Therefore, any approximation algorithm with approximation ratio greater than 5/7 can solve 3DM. The proof is now complete. □

We can now prove the inapproximability of 2/3 by refining the above analysis.

**Theorem 5** *For any fixed constant* $\epsilon$ *where* $0 < \epsilon < 1/3$, *unless* $P = NP$, *no polynomial-time algorithm can approximate MaxLoA with an approximation ratio greater* *than or equal to* $2/3 + \epsilon$, *even if all nodes (except the sink) have the same initial energy.*

*Proof* We will construct for each 3DM an equivalent instance of MaxLoA with $|M| + 2|M|n_0 + 1$ nodes, where $n_0 = \lceil \frac{1}{18\epsilon} \rceil$ and $\lceil \rceil$ is the ceiling function. This new instance can be obtained by modifying the construction in the proof of Theorem 4. To the constructed instance of MaxLoA in the previous proof, we add $2|M|(n_0 - 1)$ dummy nodes at layer 3 as follows. For each element node at layer 2, we create $n_0 - 1$ new dummy nodes at layer 3 and connect them to that node. For each dummy node at layer 2, we create $2n_0 - 2$ new dummy nodes at layer 3 and connect them to that dummy node. This construction ensures that in every round, each element node transmits at least $n_0$ messages to the sink, and each dummy node at layer 2 transmits at least $2n_0$ messages. We set $T_x = R_x = 1$ and the initial energy of every node to be $4n_0 + 1$. Since $\epsilon$ is a fixed constant, this construction can also be done in polynomial time. By similar arguments as before, we can show that 3DM has a matching if and only if the constructed instance of MaxLoA has a spanning tree with a lifetime of 1.

We now prove that any algorithm with approximation ratio $2/3 + \epsilon$ can tell whether the constructed MaxLoA instance has such a spanning tree. This is done by showing that a lifetime of 1 is equivalent to a lifetime greater than or equal to $2/3 + \epsilon$. Suppose there is a tree with a lifetime greater than or equal to $2/3 + \epsilon$. Consider each node at layer 1. Solving the inequality $\frac{4n_0+1}{2d_T(i)+1} \geq 2/3 + \epsilon$ yields $d_T(i) \leq \frac{2n_0+1/2}{2/3+\epsilon} - 1/2 < 3n_0$, where the second inequality is due to $n_0 > 1/(18\epsilon) - 1/6$. Since each subtree rooted at any node at layer 2 contains either $n_0$ nodes or $2n_0$ nodes, we have $d_T(i) \leq 2n_0$. So the spanning tree has a lifetime of 1. $\square$

## 7 Simulations

We numerically study the running time of our algorithm on simulated sensor networks. Sensors are randomly deployed in a $100 \times 100m^2$ square with initial energy uniformly drawn from $[1, 10]J$. The energy consumption for receiving a packet is $3.33 \times 10^{-4}J$ and for transmitting a packet is $6.66 \times 10^{-4}J$. Two nodes are connected to each other if their distance is no greater than the communication radius, which is 20 m. The sink node is placed at the center of the field. This setting is mainly based on [22] and [10]. We implement four algorithms in Java and run them on a laptop computer for comparison.

- ExactMaxLoA, our proposed algorithm in Algorithm 1.
- ExhaustiveSearch, a search algorithm by scanning all spanning trees. We implement the spanning tree enumeration algorithm proposed by Read and Tarjan [17], which is easy to implement, and consumes time only several times longer than other enumeration algorithms[6].
- MITT, an approximation algorithm for MaxLoA proposed by [14]. The two parameters $k_1$ and $k_2$ for controlling the number of iterations are 100 and 1000 respectively due to [14].
- MILP, a mixed ILP formulation for MaxLoA problem. We perform a binary search over all possible lifetimes[1], and for each lifetime, we formulate a mixed ILP feasibility problem by revising the formulation for capacitated spanning tree [8], followed by solving the formulated problem by LpSolve [3], a free integer linear programming solver.

### 7.1 Factors influencing the running time of our algorithm

Considering that our algorithm certainly runs slower on larger networks, we consider the impact of two other factors: (1) transmission range and (2) $T_x/R_x$, the ratio of transmission energy consumption to receiving energy consumption.

First, after generating the locations of 21 nodes, we connect two nodes if their distance is within $15, 20, \ldots, 35$ meters, leading to 5 different network topologies. We generate 50 sets of sensor locations. Figure 5(a) shows the average running time of our algorithm with respect to transmission ranges. We can see that running time roughly increases with the increase of transmission range. This is because increased transmission range leads to increased number of edges in the graph. Second, to study the impact of $T_x/R_x$, we simulate 50 networks, and for each network, we vary $T_x/R_x$ from 1 to 100 with increment of 1. Figure 5(b) shows that the average running time is relatively stable for different $T_x/R_x$. Thus, this ratio does not influence the running time much.

### 7.2 Comparison with exhaustive search

Due to the inability of ExaustiveSearch for handling large networks, we simulate a network of 21 nodes, one of which is the sink. We first generate a network,[2] and apply ExaustiveSearch to computing the optimal lifetime, followed by writing the network information, the optimal lifetime

[1] There are $n^2$ such lifetimes for a given network consisting of $n$ nodes.

[2] Only connected networks are considered

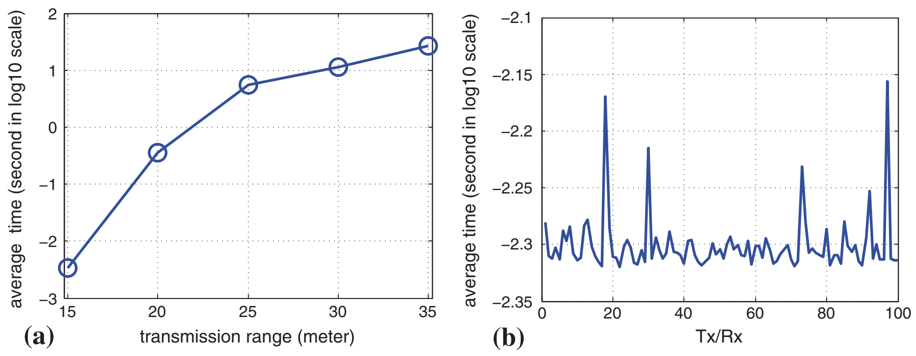

**Fig. 5** The impact of two factors on the running time of our algorithm on simulated sensor networks. **a** Impact of transmission range; **b** impact of $T_x/R_x$

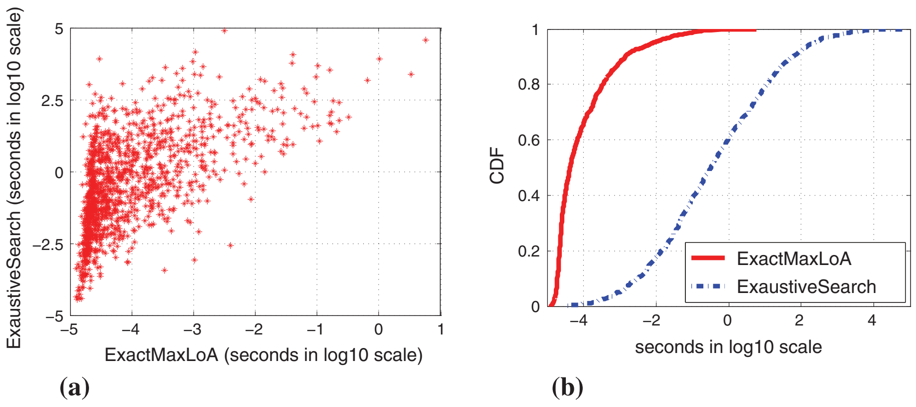

**Fig. 6** Comparing the empirical running time of our algorithm and an enumeration algorithm [17] on simulated sensor networks. **a** Scattergraph for empirical running time; **b** CDF for empirical running time

and elapsed time into a file. In situations where ExaustiveSearch cannot solve the problem in 24 h, we manually terminate the process and generate a different network. After running the program for several days, we get information for 1244 networks. These networks are also used for verifying the correctness of other implementations including that of ExactMaxLoA and MILP. This verification is necessary because, even though an algorithm is correct, a computer program implementing the algorithm may have bugs and give incorrect answers.

Figure 6 shows the running time of ExactMaxLoA against that of ExaustiveSearch on these 1244 networks. Figure 6(a) shows very long running time of ExaustiveSearch. It can take hours to compute the optimal lifetime for a network while our algorithm only takes several seconds. Additionally, it is interesting to see that greater running time of ExaustiveSearch does not necessarily imply greater running time of our algorithm. In fact, their correlation coefficient is only 0.3672. This is because, the running time of ExaustiveSearch depends on the total number of spanning trees, while that of our algorithm depends on the network structure (e.g., block structure). For the total elapsed time for these networks, ExaustiveSearch requires 67.88 h, while our algorithm needs 14.77 s. Figure 6(b) shows the CDF of the two distributions.

We then study the relationship between the elapsed time and the number of scanned trees. Figure 7 (left) shows the scattergraph for our algorithm and ExaustiveSearch. Note that the figure is in log-log scale for clarity. The points for our algorithm are roughly in a line, and the slope is 1 so that the elapsed time is linear with respect to the number of scanned trees.[3] The case for ExaustiveSearch is the same. Additionally, in terms of speed, ExaustiveSearch scans more trees than our algorithm in every second. This is because our algorithm needs to perform decomposition multiple times during scanning a single tree, increasing the required running time for each scanned tree. However, our algorithm scans much less trees than ExaustiveSearch, which is shown in the right of Fig. 7. In over 80 % networks, our algorithm scans $<0.1$ % spanning trees of that scanned by ExaustiveSearch. Consequently, our algorithm takes much less time in finding the optimal spanning tree compared to ExaustiveSearch.

## 7.3 Comparison with approximation algorithm

To compare our algorithm with MITT, we conduct two experiments. First, we apply MITT to networks recorded in Sect. 7.2 and compare its empirical running time and the

[3] Because the slope is 1, we have $\log_{10}(y) = \log_{10}(x) + b$, where $y$ is the running time, $x$ is the number of scanned trees, and $b$ is an unknown constant. Thus, $y = 10^b \cdot x$, implying $y$ is a linear function of $x$.

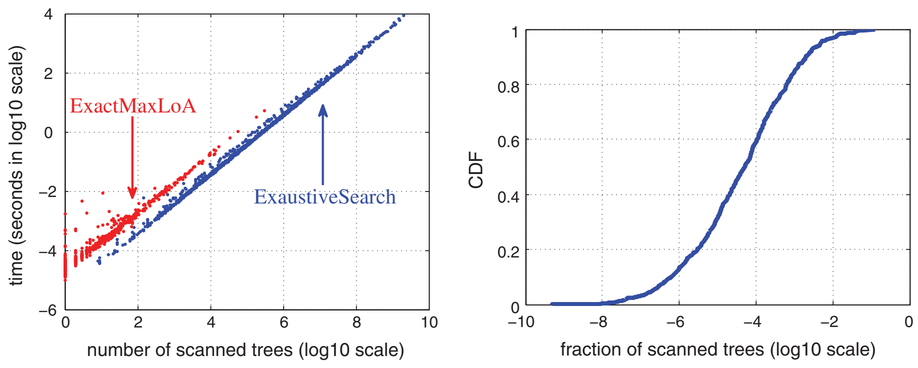

**Fig. 7** Running time vs the number of scanned trees (*left*), and the fraction of trees scanned by our algorithm (*right*)

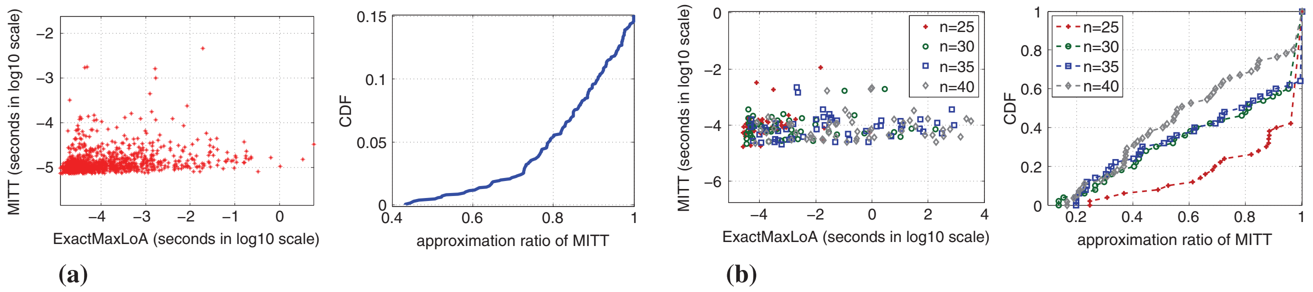

**Fig. 8** Comparing our algorithm with MITT. Approximation ratio of MITT is defined as the lifetime ratio of the tree found by MITT to the optimal tree (the tree found by our algorithm). We can see that MITT indeed runs faster, but the quality of solution may be very bad. **a** Networks in Sect. 7.2; **b** larger networks

lifetime of the obtained tree with our algorithm. The results are shown in Fig. 8(a). We can see that MITT indeed runs faster than our algorithm. For all networks, MITT terminates in 0.01 s. Unfortunately, though on most networks MITT finds trees with lifetime close to the optimal, there exist cases where MITT finds a tree with a lifetime less than half of the optimal. This coincides with the fact that MITT has poor worst-case performance guarantee.

Second, we study the impact of network size on the performance of MITT. We simulate networks with 25, 30, 35, and 40 sensor nodes by increasing the network field proportionally. In each case we simulate 50 different networks, and use MITT and our algorithm to find a spanning tree. Figure 8(b) shows that with the increase of network size, the running time of MITT remains mostly the same, while the running time of our algorithm increases noticeably. But at the same time, the larger the network is, the worse the approximation ratio of MITT becomes. This observation confirms the tradeoff between running time and the quality of solution.

## 7.4 Comparison with integer linear programming formulation

Integer linear programming is a general approach to model NP-hard optimization problems, and there are numerous free and commercial solvers for ILP problems with varying efficiency [20]. To understand its applicability to our problem, we compare the running time of MILP with that of our algorithm. The network size is varied from 25 nodes to 40 nodes with increments of 5 nodes. For each network size, we randomly generate 50 networks. During simulation, we encounter some cases where MILP does not return in several hours. Thus, to finish the simulation within reasonable time, we terminate MILP if it does not return in 1 h. Figure 9 shows the result.

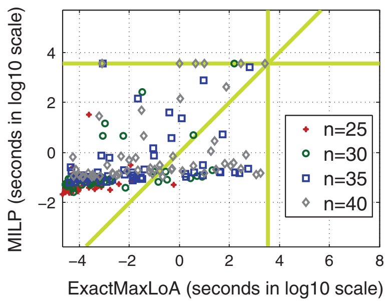

**Fig. 9** Comparing our algorithm with MILP. The three yellow lines are $y = x$, $y = \log_{10}(3600)$ and $x = \log_{10}(3600)$ respectively. There are 8 cases where MILP did not find the optimal solution within 1 h

We have two observations. First, ExactMaxLoA generally outperforms MILP. This can be noted by the fact that there are more points over the line of $y = x$. There are even eight networks where MILP failed to find the optimal solution in 1 h. Second, in some networks, MILP takes less time than MILP, meaning that these two approaches use different properties of the problem to speed up the search. It is clear that our algorithm explores the block structure of the problem and uses several greedy rules, but we are not clear what properties have been taken by MILP to occasionally achieve good running time. We believe that incorporating our algorithm with ILP solvers may further extend the capability of both algorithms.

## 8 Discussion and future work

The considered network model in this paper is simple, but it can be a starting point for more complex models. In this section, we consider the applicability of our work to other models.

Our work directly applies to the scenario where sensors have heterogeneous traffic demand, i.e., a sensor needs to transmit a number of messages to the sink in each round and this number of transmitted messages may be different for different sensors. Specifically, the inapproximability result trivially applies to this scenario, and the exact algorithm, after slight modification as mentioned in Sect. 4, can solve the corresponding problem.

Our work can also find a maximum lifetime tree for networks where every sensor consumes an additional fixed amount of energy in each round. In practice, other sensor activities may consume energy, e.g., sensing, in addition to message transmission and receiving. Previous works (e.g., [12, 14, 15, 22]) did not take such energy consumption into account due to the belief that such consumption is negligible. Considering that periodical activities such as sensing consume a fixed amount of energy in each message round, we can assume without loss of generality that each sensor node consumes additional $S$ energy in each round, besides message transmission and receiving. We refer to the lifetime maximization problem under this model as general problem. The inapproximability results for MaxLoA trivially apply to the general problem. To see that the exact algorithm in Sect. 5 also applies to the general problem, note that the lifetime of a tree $T$ in the general problem is $l(T) = \min_{i\geq 1}\frac{e_i}{d_T(i)(R_x+T_x)+T_x+S} = \min_{i\geq 1}\frac{e_i}{d_T(i)(R_x-S+T_x+S)+(T_x+S)}$. Consequently, the general problem can be reduced to MaxLoA instances with $(R_x - S)$ as the new energy consumption for receiving a message and $(T_x + S)$ as the new energy consumption for transmitting a message. The reduced instances when $R_x - S > 0$ are legitimate MaxLoA instances and can be solved by the exact algorithm. When $R_x - S < 0$, the reduced instances are not legitimate MaxLoA instances because the new energy consumption for receiving a message is not positive. Fortunately, one can check that the exact algorithm can also solve these instances since all the facts, rules and theorems in Sect. 4 still hold for them.

In addition, sensor networks operating in low duty cycle mode exhibit similar energy consumption properties as our model in that transmitting/receiving a message consumes a fixed amount of energy [16]. Therefore, as long as a single routing tree is required, the maximum lifetime data gathering tree problem for such networks is the same as our problem.

In the future, we plan to extend our work to deal with topology changes. In practice, the topology of a network may change in three cases: (1) an existing link disappears due to obstacles; (2) a new link appears due to removal of obstacles; (3) a node dies. In all cases, a naive scheme is to re-execute our exact algorithm, but we can see that there are redundant subproblems across the change. We plan to study the problem structure across the change and speed up the process of finding a new optimal tree. In addition, we plan to investigate the possibility of incorporating our algorithm with ILP formulation. As suggested by the simulation results, this direction is promising.

## 9 Conclusions

In this paper, we prove that no polynomial-time algorithm can guarantee to find a spanning tree with lifetime greater than 2/3 of the maximum lifetime for sensor networks without aggregation, unless $P = NP$. In addition, we propose an algorithm to solve the problem exactly. We prove that the algorithm is correct and runs in $O(mn)$ space where $m$ is the number of edges and $n$ is the number of vertices. This algorithm runs in exponential time in the worst case, but is much faster than enumerating all spanning trees. Simulations show that, for networks where enumerating all spanning trees requires several hours, our algorithm takes only a few seconds.

**Acknowledgments** This work was partly supported by the New-Faculty Research Grant of Nanjing University of Aeronautics and Astronautics, China NSF Grants (61321491, 61373130, 61133006), NSF of Jiangsu Province Grant (BK20141319), and China 973 projects (2014CB340303, 2012CB316201).

## References

1. Altinkemer, K., & Gavish, B. (1988). Heuristics with constant error guarantees for the design of tree networks. *Management Science*, *34*(3), 331–341.
2. Arkin, E. M., Guttmann-Beck, N., & Hassin, R. (2012). The (k, k)-capacitated spanning tree problem. *Discrete Optimization*, *9*(4), 258–266.
3. Berkelaar, M., Eikland, K., & Notebaert, P. (2010). lp_solve 5.5.2.0. http://lpsolve.sourceforge.net/5.5/. Accessed 2014-3-2.

4. Cormen, T. H., Stein, C., Rivest, R. L., & Leiserson, C. E. (2001). *Introduction to algorithms* (2nd ed.). New York: McGraw-Hill Higher Education.
5. Diestel, R. (2006). *Graph theory* (3rd ed.). Berlin: Springer.
6. Gabow, H. N., & Myers, E. W. (1978). Finding all spanning trees of directed and undirected graphs. *SIAM Journal on Computing*, *7*(3), 280–287.
7. Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-completeness*. San Francisco: W. H. Freeman & Co.
8. Garey, M. R., & Johnson, D. S. (1983). Formulations and algorithms for the capacitated minimal directed tree problem. *Journal of the ACM*, *30*(1), 118–132. doi:10.1145/322358.322367.
9. Hopcroft, J., & Tarjan, R. (1973). Algorithm 447: Efficient algorithms for graph manipulation. *Communications of ACM*, *16*(6), 372–378.
10. Intanagonwiwat, C., Govindan, R., & Estrin, D. (2000). Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of ACM MobiCom*.
11. Jothi, R., & Raghavachari, B. (2005). Approximation algorithms for the capacitated minimum spanning tree problem and its variants in network design. *ACM Transations on Algorithms*, *1*(2), 265–282.
12. Kuo, T. W., & Tsai, M. J. (2012). On the construction of data aggregation tree with minimum energy cost in wireless sensor networks: Np-completeness and approximation algorithms. In *Proceedings of IEEE INFOCOM*.
13. Lenstra, J. K., Shmoys, D. B., & Tardos, E. (1990). Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, *46*(3), 259–271.
14. Liang, J., Wang, J., Cao, J., Chen, J., & Lu, M. (2010). An efficient algorithm for constructing maximum lifetime tree for data gathering without aggregation in wireless sensor networks. In *Proceedings of IEEE INFOCOM*.
15. Luo, D., Zhu, X., Wu, X., & Chen, G. (2011). Maximizing lifetime for the shortest path aggregation tree in wireless sensor networks. In *Proceedings of IEEE INFOCOM*.
16. Peng, Y., Li, Z., Qiao, D., & Zhang, W. (2013). I2c: A holistic approach to prolong the sensor network lifetime. In *Proceedings of IEEE INFOCOM*.
17. Read, R. C., & Tarjan, R. (1975). Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, *5*, 237–252.
18. Shan, M., Chen, G., Luo, D., Zhu, X., & Wu, X. (2014). Building optimal shortest path data aggregation trees in wireless sensor networks. *ACM Transactions on Sensor Networks, 11*(1). Article No. 11.
19. Shioura, A., Tamura, A., & Uno, T. (1997). An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM Journal on Computing*, *26*(3), 678–692.
20. Wikipedia Contributors. (2014). *Linear programming*. http://en.wikipedia.org/wiki/Linear_programming. Accessed 2014-3-2.
21. Williamson, D. P., & Shmoys, D. B. (2011). *The design of approximation algorithms* (1st ed.). New York, NY: Cambridge University Press.
22. Wu, Y., Mao, Z., Fahmy, S., & Shroff, N. (2010). Constructing maximum-lifetime data-gathering forests in sensor networks. *IEEE/ACM Transactions on Networking*, *18*(5), 1571–1584.
23. Xiang, L., Luo, J., & Rosenberg, C. (2013). Compressed data aggregation: Energy efficient and high fidelity data collection. *IEEE/ACM Transactions on Networking, 21*(6), 1722–1735.

**Xiaojun Zhu** is an Assistant Professor at the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. He obtained his BS degree in Computer Science from Nanjing University in 2008 and got his PhD degree from Nanjing University in 2014. From August 2011 to August 2012, he was a Visiting Scholar at the College of William and Mary. His research interests include wireless sensor networks, vehicular networks, RFID system, smartphone system, and cognitive radio networks.

**Xiaobing Wu** is an Associate Professor in Nanjing University. He received his BS and MEng degrees in Computer Science from Wuhan University in 2000 and 2003, respectively. He earned his PhD degree in Computer Science from Nanjing University (NJU) in 2009. He had been with China Merchants Bank for two years before moving to NJU in 2005. His research interests are mainly in the areas of wireless networking.

**Guihai Chen** obtained his BS degree from Nanjing University, M. Engineering from Southeast University, and PhD from University of Hong Kong. He visited Kyushu Institute of Technology, Japan in 1998 as a research fellow, and University of Queensland, Australia in 2000 as a visiting Professor. During September 2001 to August 2003, he was a visiting Professor in Wayne State University. He is a Distinguished Professor with the Department of Computer Science, Shanghai Jiao Tong University. Prof. Chen has published more than 280 papers in peer-reviewed journals and refereed conference proceedings in the areas of wireless sensor networks, high-performance computer architecture, peer-to-peer computing and performance evaluation. He has also served on technical program committees of numerous international conferences.