An Extended Link Reversal Protocol in Dynamic Networks

Jie Wu and Fei Dai
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431

***Abstract*—** We consider the problem of maintaining routing paths between nodes in a dynamic network. Gafni and Bertsekas proposed a *link reversal* approach called the BG method that maintains a directed acyclic graph (DAG) with a given destination as the sink node. By virtue of built-in redundancy, an updating algorithm to establish a new DAG is activated infrequently and it happens only when the last outgoing link of a host in the DAG is destroyed due to the movement of nodes. In this paper, we propose another updating approach that tries to minimize the total number of reversed links and to maintain routing information without using much extra overhead. The approach maintains a reversed breadth-first tree. Nodes in the network are either marked (inside the tree) or unmarked (outside the tree). When it is too costly to maintain a minimum path for a marked node, the branch rooted at the node is trimmed and the approach then gracefully switches to the BG method. Several extensions are also discussed. A simulation study is conducted to compare the performance of the proposed approach with the existing one. [1]

1. Introduction

Consider a directed acyclic graph (DAG) with a special node called a destination node, the DAG is *destination oriented* if for every node there exists a directed path originating at this node and terminating at the destination node; otherwise, it is called *destination disoriented*. A DAG is destination disoriented if and only if there exists a node other than the destination which is a sink node with no outgoing link. Gafni and Bertsekas considered the following problem [3]: Given a connected destination disoriented DAG, transform it to a destination oriented DAG by reversing the directions of some of its links.

An application of the above problem is routing in dynamic networks which include ad hoc networks [4] and sensor networks [2]. Here we focus on a special type of dynamic network where the network topology changes via node failure/recovery and link switching-on/-off. When network topology changes over time, *a DAG for a given destination does not need to be changed as long as each node has a downstream neighbor*. That is, the update of the DAG can be postponed until a host loses its last outgoing link and, then, an updating algorithm is invoked to make the DAG destination oriented again.

Gafni and Bertsekas [3] proposed two updating algorithms, simply called the GB method, based on reversing the directions of certain links in the DAG: *full reversal* and *partial reversal*. It has been proved in [3] that both algorithms terminate in a finite number of steps and the resultant graph is still a destination oriented DAG. The GB method has been adopted in the routing protocol TORA [6] for ad hoc networks.

[1]This work was supported in part by NSF grant CCR 9900646 and grant ANI 0073736. Email:{jie,fdai}@cse.fau.edu.

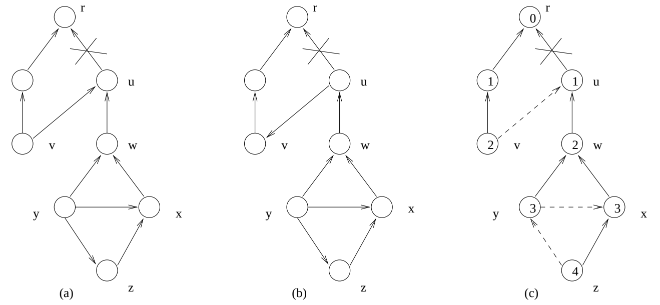

Fig. 1. A destination oriented DAG with (a) an isolated branch rooted at w, (b) DAG after partial link reversal, and (c) a spanning reversed breadth-first tree (RBFT).

Although both updating algorithms in the GB method have several elegant features, they do not maintain a path of minimum number of hops (or simply *minimum path*) for a node to the destination. The GB method does provide a "watermark" for each node and it is used to determine the orientation of each link. A greedy routing approach can be adopted that selects a neighbor with the lowest watermark at each routing step. However, the watermark of each node does not reflect the distance to the destination and, hence, the routing process is unpredictable. In general, maintaining a minimum path for each node in dynamic networks is a challenging problem. In this paper, we propose another updating approach which tries to maintain routing information in a given destination oriented DAG without using much extra overhead. The destination oriented DAG maintains a *reversed breadth-first tree* (RBFT); that is, it contains a minimum path from each node in the tree to the destination (also called the root node) in the dynamic network.

A related work in this field is to maintain a *spanning shortest path tree* (SPT) in a given graph based on a different graph model. When topology and/or link costs of the network change, either the existing SPT is re-constructed or updated [5]. However, such an update has to be invoked whenever there is a change, making it too costly for practical use.

RBFT differs from SPT in two aspects: (1) RBFT is a minimum tree (in terms of hop count) while SPT is a shortest path tree. Hop count is a commonly used measure in ad hoc networks. (2) SPT is a spanning tree while RBFT may or may not be a spanning tree of the graph. In general, nodes in the network are divided into *marked* nodes (inside the RBFT)

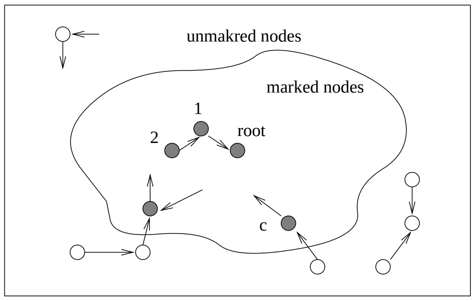

Fig. 2. Marked and unmarked nodes in a destination oriented DAG.

and *unmarked* nodes (outside the RBFT). A *level* indicating the distance to the root is associated with each marked node. Unmarked nodes do not have distance information. Initially, all nodes are marked nodes upon the construction of a spanning RBFT and, hence, a destination oriented DAG is constructed where the direction of a link is directed from a higher level node to a lower level node (node id is used to break a tie when two end nodes have the same level). Figure 1 (c) shows a spanning RBFT. When a node loses its outgoing tree link (such as node u in Figure 1 (c)), the proposed approach adjusts the level of the node and those of its descendants. When it is too costly to maintain a minimum path for a node w by keeping its level up-to-date, the branch rooted at w is *trimmed*; that is, node w and all its descendants are unmarked. The approach is then gracefully switched to either full or partial reversal. When a new node is added to the network, it can be marked or unmarked depending on the marked/unmarked status of its neighbors.

Figure 2 shows the general structure of marked and unmarked nodes. The directions of links connecting marked nodes are determined by RBFT. An unmarked node always points to its marked neighbors (if any). The direction of a link between two unmarked nodes is decided based on the execution of the GB method. When an updating algorithm is initiated at an unmarked node, the GB method is used. However, it will not affect the existing RBFT structure. When an updating algorithm is initiated at a marked node, the proposed approach is applied, but it may switch to the GB method after a branch of an RBFT is trimmed. Therefore, the GB method and our approach co-exist, and a destination oriented DAG is always maintained. During the course of updating, the marked node set shrinks while the unmarked node set grows. Unmarked nodes can also be selectively remarked in a careful way to avoid routing loop. If maintaining the minimum path property for every node is required or the marked node set becomes too small, the current destination oriented DAG is flushed and a new spanning RBFT is constructed (and hence the new DAG) through a global flooding.

Throughout the paper, we assume: (1) Each link is bidirectional and the graph is connected, unweighted, and undirected. (2) The graph structure is dynamic as a result of link failure/recovery and/or node switching on/off (a frequent operation in a sensor network). (3) Each node knows its neighbors; that is, it always keeps its neighbor set up-to-date. (4) There are sufficient routing requests in each interval of two adjacent graph structural changes.

2. PRELIMINARIES

The GB Method. The GB method [3] includes two algorithms: (1) *Full reversal*: At each iteration each node other than the destination that has no outgoing link reverses the directions of all its incoming links. (2) *Partial reversal*: Every node u other than the destination keeps a list of its neighboring nodes v that have reversed the direction of the corresponding link (u, v)[2]. At each iteration each node u that has no outgoing link reverses the directions of the links (u, v) for all v which do not appear on its list, and empties the list. If no such v exists, node u reverses the directions of all incoming links and empties the list.

Both reversals can be implemented by assigning "watermark" for each node and raising the watermark of a node based on the above updating algorithms. The orientation of each link is determined by the watermarks of two end nodes. The destination always has the lowest watermark. Therefore, the link reversal algorithms can be carried out by raising the watermark of each sink node (other than destination) until there is only one sink node left which is the destination. Although both updating algorithms have several elegant features, they do not maintain a minimum path for a node to the destination. Besides, the total number of links reversed is not minimal.

For partial reversal, the *isolated branch problem* exists. Let $RT(w)$ be a set of nodes that are reachable to node w. A subgraph containing w is called an *isolated branch* if (1) w has only one outgoing link and (2) any node in $RT(w)$ connects only nodes in $RT(w)$. In Figure 1 (a), the branch rooted at w is an isolated branch. When link (u, r) is broken, links (u, v) and (u, w) are reversed in both full reversal and partial reversal. In full reversal, the sequence of nodes that are involved in reversal are: $w \rightarrow x \rightarrow z \rightarrow y$. When a node is involved in full reversal all its adjacent links reverse their directions. When partial reversal is applied, adjacent links that are reversed in the early iterations will not reverse their links again in the current iteration. Each node has a list of adjacent nodes that have reversed the directions of the corresponding links. For example, at node w due to the fact that there is no outgoing link from w, w is revolved in the reversal and it will reverse links (w, x) and (w, y), but not link (u, w) since it has been reversed. Note that the above exemption (for link (u, w)) is only good once. The subtle point is in resetting the list once partial reversal is applied at the corresponding node. That is, after the link reversal is applied at node u, node u erases its memory of its past list. Consequently, the sequence of nodes that are involved in reversals is: $w \rightarrow x \rightarrow z \rightarrow y \rightarrow z \rightarrow x \rightarrow w$. The sequence of links that have been reversed in the corresponding reversals is: $\{(w, x), (w, y)\} \rightarrow \{(x, y), (x, z)\} \rightarrow \{(y, z)\} \rightarrow \{(w, y), (x, y), (y, z)\} \rightarrow \{(x, z)\} \rightarrow \{(w, x)\}$

[2] (u, v) represents an undirected link between u and v.

$\rightarrow \{(u, w)\}$. In this case, not only do all the links in the isolated branch reverse their directions *twice* (see Figure 1 (b)), but also all nodes (except node y) apply partial reversal twice!

Reversed Breadth-First Trees. In the GB method, a destination oriented DAG is constructed from a given connected, unweighted, and undirected graph by assigning a direction for each link. One way to maintain the DAG is by generating a reversed spanning tree rooted at the destination (root). Each node u is assigned a *level* $L(u)$ such that for any link (u, v) in the tree, $L(u) > L(v)$ if and only if the link is directed from u to v. Any link (u, v) not in the tree is directed from u to v if and only if $L(u) > L(v)$ or $id(u) > id(v)$ when $L(u) = L(v)$, where each node has a distinct node id. The orientation of its adjacent links can be directly derived by comparing the levels of two end nodes. Basically, $L(u)$ can be considered a watermark for node u. Water always flows from a higher mark to a lower mark. The destination has a lowest watermark. However, $L(u)$ does not provide useful routing information such as the distance of u to the destination.

In our approach, we tie $L(u)$ to the distance between u and the destination. We use the term level to represent a watermark that is tied to the distance; otherwise, it is still called a watermark. The destination-oriented DAG is constructed by finding a spanning *reversed breadth-first tree* (RBFT) rooted at the destination. A tree is called breadth-first if each node at distance d from the root appears at depth d in the tree. In fact, each node is connected to the root through a minimum path (in terms of hop count). All the links in the networks are classified into *tree links* and *non-tree links*. If a tree link is directed from u to v, then u is called a child node of v (and v the parent node of u). Let level L be defined in such a way that $L(u) = L(v) + 1$ where u is the child of v. Clearly, $L(u)$ is precisely the distance of u to the root. Figure 1 (c) shows a DAG generated from a spanning RBFT. In Figure 1 (c) the number inside each node is the level of the node (i.e., the distance to the next node). Solid lines are tree-links and dashed links are non tree-links. Note that not all DAGs can be generated from a spanning RBFT. For example, Figure 1 (a) cannot be generated from any spanning RBFT.

Problems. We try to maintain an RBFT (not necessarily a spanning one) for a given destination in a dynamic network. Nodes in the RBFT are called *marked nodes* with assigned levels. Nodes outside the RBFT are called *unmarked nodes*. Still each unmarked node keeps its watermark used to determine the orientations of its adjacent links. The watermark of an unmarked node is higher than that of a marked one. The destination still has the lowest watermark. Two requirements are given: (1) *Destination Oriented DAG Requirement* (DAGR): The network as a whole (marked nodes and unmarked nodes) should be maintained as a destination oriented DAG by assigning a direction for each link. (2) *Level Requirement* (LR): The level associated with each marked node u corresponds to the distance between u and the destination in the subnetwork induced by marked nodes. In addition, the set of marked nodes (which is dynamic) should be kept as large as possible so long as no significant overhead is introduced in maintaining such a set.

It is possible that we require a Strong Level Requirement (SLR): The level associated with each marked node u corresponds to the distance between u and the destination in the *whole network*. Clearly, SLR covers LR. However, SLR is too strong and is difficult to enforce. In the subsequent discussion, we only consider LR. In [9], two weak level requirements are discussed: Weak Level Requirement (WLR) and Upper-Bounded Level Requirement (ULR).

The topology of a dynamic network can be changed by three primitive operations: *delete-a-link*, *add-a-link*, and *add-a-node*. The delete-a-node can be considered as several delete-a-link operations. Node switching on/off can be implemented as add-a-node/delete-a-node. Although node movement (a typical operation in ad hoc wireless networks) is not considered here, it can be viewed as a sequence of delete-a-node/delete-a-link and add-a-node/add-a-link operations.

3. Proposed Approach

Basic Ideas. We first establish a spanning RBFT in the network to meet both DAGR and LR. Then we try to maintain DAGR and LR as invariants upon a sequence of delete-a-link, add-a-link and add-a-node operations.

It is relatively easy to handle an add-a-node operation. The new node can be marked or unmarked depending on the marked/unmarked status of its neighbors. A safe and easy way to ensure both DAGR and LR is put an unmarked status on the new node and all adjacent nodes are directed away from the node. However, the requirement for a large set of marked nodes needs to mark the new node whenever possible. For an add-a-link operation, the direction of the link is decided based on the levels of two end nodes. The branch rooted at an end node is trimmed if the levels of two end nodes differ by more than 1.

The way to handle a delete-a-link operation is more involved, especially when it is an outgoing link of a marked node. When the deleted link is the outgoing link of an unmarked node, the GB method is directly applied. Our approach is applied when the deleted link is the outgoing link of a marked node. Here we introduce some basic concepts. A neighbor of a marked node can be classified into *marked/unmarked* and *child node/non-child node*. A *disconnected branch* rooted at u is a disconnected branch of an RBFT as a result of the delete-a-link operation. That is, the link that is deleted is the outgoing tree link of u (see Figure 1 (c)).

A *replacement* v for node u's parent node is a marked neighbor that has the same level as the level of u's parent (i.e., $L(v) = L(u) - 1$). When replacement for u's parent node is found, since the level of node u remains unchanged, the levels of its descendants remain the same. A *substitute* v for node u is a non-child marked neighbor (but not a replacement) that has the lowest level; that is, $L(v) \leq L(u)+1$ based on the level definition. Note that there are only two possibilities for v and u: either $L(v) = L(u)$ or $L(v) = L(u) + 1$. When a substitute

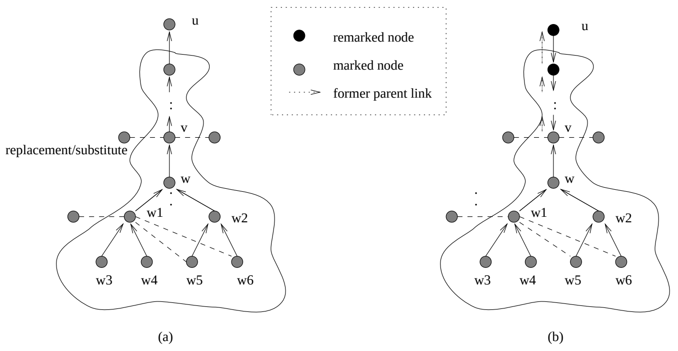

Fig. 3. A disconnected branch rooted at node u: (a) before the updating process (b) a former parent chain, a successful replacement/substitute at v, and a multiple-child node w.

for u's parent node is used, since the level of u increases, the levels of its descendants also need to be adjusted.

The basic idea is to update the parent of each node in a disconnected branch using replacement/substitute following the branch down the tree. The updating process either completes successfully or switches to the GB method when it is too costly to keep the level of a node in the branch up-to-date. The updating process terminates successfully whenever a replacement is found or all nodes in the branch have their substitutes. When a node cannot be updated by either replacement or substitute, it is labelled as *remarked*, and the link to its child node is reversed if it is a single-child node. All remarked nodes are linearly chained through *former parent links* (which are reverses of regular links). When a descendant is successfully updated by a replacement/substitute, levels of nodes in the former parent chain (if any) can be remarked by incrementing the level by one from node to node until reaching node u. When a descendant is successfully replaced/substituted, single-child descendants can be at least substituted. It is rather difficult to come up with a simple solution for the multiple-child node case, since it has multiple branches. Each branch of a multiple-child node will be trimmed unless a replacement is found.

Figure 3 shows a disconnected branch rooted at node u. Figure 3 (b) shows a former parent chain which is always a "prefix" of the disconnected branch. A successful replacement/substitute occurs at node v, and levels of nodes in the chain are remarked accordingly. If a replacement is found at node v, the updating process completes successfully. If it is a substitute at node v, the updating process continues at descendants of v. Node w is a multiple-child node. Depending on the frequency of various operations the set of marked nodes reduces over the time. When the set becomes empty, the proposed method is degraded to the BG method. In this case, a global flooding is needed to construct a new spanning RBFT and all nodes are remarked again.

RBFT Initialization. RBFT can be easily constructed in a synchronous network. Level information is piggybacked with a *search* message which is passed level from level, one level per round, from the destination node. In an asynchronous network, messages travel at different speeds, so erroneous parent designations may happen but they can be corrected as follows: If node u initially identifies one of its neighbors, say v, as its parent, and later obtains information from another neighbor w along a shorter path, node u can change its parent designation to w. In this case, node u must inform its other neighbors about its correction, so that they might also correct theirs. The Dijkstra and Scholten's diffusing algorithm [1] for the termination-detection problem can be used so that each node knows when to stop the process.

RBFT Update – add-a-node. To simplify the discussion, we assume that all status such as marked/unmarked and levels are stable when a delete-a-link, add-a-link, or add-a-node operation is applied. A level of marked node u is *stable* if it has a correct level. In fact, when marked node u is stable $|L(u)-L(v)| \leq 1$ for each marked neighbor v of u. A network is stable if all marked nodes have the correct levels. First of all, let's consider a safe and easy way to add a node which ensures both DAGR and LR:

- *safe-add*: Unmark the new node and all adjacent links are directed away from the node.

However, due to the requirement for a large set of marked nodes, the new node needs to be marked whenever possible. The following approach can be used:

- *best-add*: If levels of all marked nodes differ by no more than 2 then level $L+1$ is assigned, where L is the minimum level of neighbors; otherwise, apply safe-add.

When the levels of marked neighbors differ by no more than 2, the newly added node will not make a neighbor level unstable. When the new node is assigned $L+1$ (L is the minimum level among them), its level is stable and levels of all other nodes (including neighbors) are stable. Therefore, the assignment is correct. Note that when the levels of marked neighbors differ by more than 2, an unstable level occurs when the new node is assigned $L+1$. Note that stability can be reached by iteratively adjusting levels of unstable neighbors. This is an expensive process since the adjustment is not limited to the one branch. Another option, as used in add-a-link, is to trim branches rooted at unstable neighbors. However, this approach will greatly reduce the size of the marked node set, therefore, the safe-node procedure is adopted.

Consider adding a new node to Figure 1 which is adjacent to nodes r, u, v and w. Since the levels of these nodes differ by at most 2, the new node is assigned a level of 1. When a new node is adjacent to node r and node x, the new node is unmarked and adjacent links are directed toward neighbors.

RBFT Update – add-a-link. When one of the end nodes of the added link (u, v) is unmarked, the direction of the link is determined by the watermarks/levels of u and v. When both end nodes are marked and their levels differ by no more than 1, its direction is decided by the levels and id's of u and v. When the levels of two end nodes differ by more than 1, the branch rooted at one end node will be trimmed. Note that the adjustment approach can also be applied; however, this process is too expensive since it is not limited to one branch.

```
add-a-link(u,v): /* upon adding link $(u,v)$ */
  1) If both $u$ and $v$ are marked nodes and their levels differ by
     no more than 1, or, one is marked and the other is unmarked,
     link $(u,v)$ is added and its direction is decided by the levels
     and ids of $u$ and $v$.
  2) If both $u$ and $v$ are marked nodes but their levels differ by
     more than 1, then the branch rooted at the end node with a
     higher level is trimmed.
  3) If both $u$ and $v$ are unmarked nodes, the direction of the link
     is determined by the watermarks of $u$ and $v$.
```

```
delete-a-link(u): /* upon removal of an outgoing link of node $u$ */
  1) If $u$ is unmarked, call the GB method which is either full or
     partial reversal.
  2) If $u$ is marked, exit when the outgoing link is not a tree link;
     otherwise, call update-level(u).
```

When a node trims its branch, it sends an unmark signal to all its descendants (including remarked nodes in the former parent chain). When a node receives an unmark signal it changes its status to unmarked and forwards the signal to its child nodes. In the example of Figure 1 (c), if *add-a-link(r,w)* is called, the branch rooted at w will be trimmed. Note that during the trimming process, when a marked node is unmarked the directions of its adjacent links remain unchanged. In Figure 1 (c), if *add-a-link(v,x)* is called, link (v,x) is added and the direction of the link is pointed from x to v. If *add-a-link(v,z)* is called, link (v,z) is added but node z is unmarked.

RBFT Update – delete-a-link. Operation *delete-a-link(u)* considers two cases: When u is marked and when u is unmarked. The GB method is used when u is unmarked. The rest of discussion focuses only on the case when u is marked. When a *delete-a-link(v)* removes an adjacent link of node u, nothing else needs to be done if this link is not an outgoing tree link; otherwise, node u as the initiator calls *update-level(u)*. Procedure *update-level(v)* is for either u or a descendant v of u. When v is a multiple-child node without a replacement, *multiple-child-node(v)* is called. Assume that initially the former parent chain is empty.

When a node receives a network partition warning but has an unmarked neighbor, it erases the warning signal and stops; otherwise, it forwards the warning signal to its former parent. When the node is the initiator and there is no adjacent unmarked neighbor, a network partition is detected. Note that the detection of a network partition is limited to only the subgraph induced by the marked nodes.

In Figure 3 the 2-hop descendant set of w is $\{w_1, w_2, w_3, w_4, w_5, w_6\}$. Such a set is required to prevent a child node (say w_1) from selecting a descendant of w (say w_6) as its replacement/substitute. At a multiple-child node, say w in Figure 3, if a replacement is found for w's parent, nothing else needs to be done; otherwise, the root of each branch (say w_1 in the figure) tries to find a replacement/substitute, the corresponding branch is saved only if a replacement is found; otherwise, the branch (excluding the root w_1) is trimmed.

```
update-level(v):
  1) If a replacement exists, replace $v$'s parent, adjust the former
     parent chain (if any), and exit.
  2) If a substitute exists, substitute $v$'s parent and adjust the former
     parent chain (if any).
       a) If $v$ is single-child node, $v$ sends an update level signal
          to its child node (if any), and exit.
       b) If $v$ is multiple-child node, call multiple-child-node(v).
  3) If neither replacement nor substitute exists, then
       a) If $v$ is a multiple-child node, $v$ is remarked and call
          multiple-child-node(v).
       b) If $v$ is a single-child node, $v$ reverses the link $(v,w)$
          where $w$ is the child node ($v$ is called the former parent
          node of $w$), and $v$ becomes a remarked node and then it
          sends an update level signal to $w$.
       c) If $v$ has no child node, $v$ sends a "network partition warn-
          ing" signal to the initiator $u$ via the former parent chain.
          When a node receives a "network partition warning" but
          has an unmarked neighbor, it erases the warning signal
          and stop; otherwise, it forwards the warning signal to its
          former parent. When the node is the initiator $u$ and there
          is no adjacent unmarked neighbor, a network partition is
          detected.
```

RBFT Update – enlarge-the-set. If an unmarked node has a marked neighbor, it can be marked again. This process resembles adding a new node. When an unmarked node u has a marked neighbor and the levels of marked neighbors differ by no more than 2, and suppose L is the minimum level among neighbors, node u can be remarked to level $L+1$. This process also needs to be controlled; otherwise, the remarking process resembles a global flooding.

4. Examples and Properties

Consider the example shown in Figure 4 (a). Assume that an RBFT has been constructed and the level of each node is also shown in Figure 4 (a). Suppose link (c,e) is removed, node e finds a replacement b for c (since their levels are the same) and nothing else needs to be done. When link (e,h) is broken, node h cannot find a replacement. Since h is a multiple-child node, it sends its 2-hop descendants set $\{k,l,n\}$ to its child nodes k and l, k finds its replacement g but l does not. Node h adjusts its level based on the level of k and the level of l in turn is based on that of h. Node l trims its branch consisting of node n only. When link (f,j) is removed, j cannot find a replacement/substitute, nodes j and m form a former parent chain. Node m finds a replacement i and m adjusts the level of j through the former parent chain.

When two new nodes o and p are added in Figure 4 (b), since the levels of o's neighbors differ by more than 2, node o is unmarked. The levels of p's neighbors differ by no more than 2, so node p is assigned a level of 3. During the enlarge-the-set process, node n is remarked to 5 since the levels of its neighbors differ by no more than 2.

In Figure 1 (c), when link (r,u) is removed, node u finds a substitute node v as its parent node, and the substitute for w's parent is still node u (but with a new level). Since node w is a multiple-child node and neither node y nor node x

multiple-child-node(v):

1) Node v forwards its 2-hop descendent set to its child nodes so that each child knows its neighbors that are descendants of v.
2) Each child node finds a replacement/substitute that is not in the 2-hop descendent set forwarded from v and sends back its level to v if such a replacement/substitute is found.
3) If v is a remarked node and no adjustment is received from its child nodes, v trims its branch (including v), and exit; otherwise, v adjusts its level based on the levels of all its neighbors. Then for each child node, if a replacement is found, nothing else needs to be done; otherwise, v adjusts its level based on all its neighbors (excluding its child nodes) and trims its corresponding branch (excluding itself).

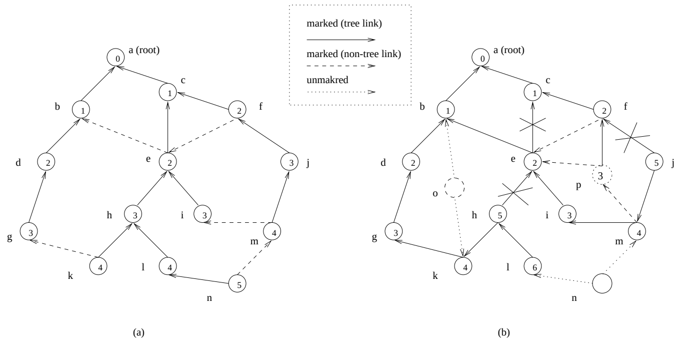

Fig. 4. An example.

has a replacement, node z is trimmed. The adjusted levels for nodes u, w, y and x are 3, 4, 5 and 5, respectively. Node z is re-assigned a level of 6 after the application of enlarge-the-set. We say a node is *affected* by an adjustment if either its adjacent tree links are changed or the direction of its adjacent links are reversed. The following theorems show our major results. The details of proofs can be found in [9].

Theorem 1: *When the GB method is applied to an unmarked node, only unmarked nodes are affected.*

Clearly, both DAGR and LR requirements are ensured if the GB method is applied to an unmarked node. A *boundary node* is an unmarked node with at least one adjacent marked node. Clearly, a boundary node will not be affected by the GB method since it has at least one outgoing link to a marked node. Therefore, no marked node will be affected if the GB method is applied to an unmarked node. DAGR is ensured by the GB method. LR is also ensured since marked nodes are unaffected.

Theorem 2: *The proposed approach ensures both DAGR and LR requirements when it is applied to a marked node.*

The ways replacement and substitute are defined ensure the LR requirement and the DAGR requirement within the marked node set. When branches of an RBFT are trimmed, their link directions remain unchanged. Therefore, DAGR is still maintained within the unmarked node set. Based on Theorems 1 and 2, both DAGR and LR requirements are preserved in the network (which includes marked and unmarked nodes).

The loop freedom is ensured if all operations (including enlarge-the-set) occur in a sequential order and the next operation occurs after the network is stabilized after the current operation. Otherwise, routing loops will occur like in any routing protocols where the distance information of a node is dependant on the distance information stored in its neighbors. The occurrences of loops depend on how fast various level updates and trimming processes can be carried out. If the propagation is relatively slow, a loop is likely to occur. For example, the count-to-infinity problem may occur where a node has raised its level several times before all its descendants were able to update theirs. The node may end up selecting one of its descendants as its parent node. The loop freedom can be enforced in one of the following three ways: (1) Support only *replacement* with no *substitute*: loop freedom is ensured if no node can raise its level and, hence, avoid the count-to-infinity problem. (2) Support sequence numbers: Each level is associated with a sequence number issued by the destination. A replacement (substitute) operation can be issued if the corresponding node has a higher or equal (higher) sequence number. (3) Lock all descendants: A replacement/substitute operation is performed after a confirmation is received from each child node. This is an iterative process that also applies to each child node. In this way, a replacement (substitute) operation is performed only after all descendants have completed their updates. This approach was first proposed in [7], [8].

5. Performance Study

The performance and overhead of the proposed approach is evaluated and compared with other approaches via simulation. Four routing algorithms are evaluated, all of them based on the idea of maintaining a destination oriented DAG, including Gafni and Bertsekas' full reversal (FR) and partial reversal (PR) methods, the proposed method based on RBFT with the constraint that only replacement is supported (RO), and the proposed method without any constraint, supporting both replacement and substitute (RS). For all four algorithms we evaluate their performance in terms of the average length of a routing path (in terms of hop count) and their message and time costs. For the two RBFT-based algorithms (i.e., RO and RS) we also evaluate their shrinking speeds of the RBFT, which is useful in determining the interval between two calls for global flooding.

Simulations are conducted using a discrete-event simulator. To generate a random network, n nodes are randomly placed in a confined 100×100 area. A link is inserted between two nodes if their distance is smaller than a given transmission range r. To achieve a given density, r is adjusted based on an average node degree d to produce exactly $nd/2$ links in the network. In the beginning of a simulation, one node is designated as the destination, and a destination oriented DAG is established by each algorithm. For FR and PR, the destination oriented DAG is established by applying link reversal rules, and for RO and RS, by a global flooding. Link faults are simulated by removing randomly chosen links. Because network partition cannot be properly handled by the GB methods, only links in cycles can be removed. The maintenance operation triggered

TABLE I

PERFORMANCE AND OVERHEAD OF FOUR LINK REVERSAL ALGORITHMS.

	Sparse Network				**Dense Network**			
	Routing Len.		Overhead		Routing Len.		Overhead	
Algo.	RND	WM	Step	Msg	RND	WM	Step	Msg
FR	9.31	7.51	0.64	1.23	5.19	3.32	0.04	0.05
PR	9.20	7.44	0.66	1.46	5.19	3.32	0.04	0.05
RO	7.71	7.07	0.84	1.42	2.84	2.69	0.20	0.28
RS	7.39	7.02	1.05	2.06	2.72	2.68	0.27	0.39

by a link fault is simulated in a synchronized manner. Each maintenance operation takes several steps. Control messages are exchanged between neighbors, and each control message takes exactly one step to be generated, sent and received by neighbors. We assume that the interval between two link faults are long enough so that every maintenance operation ends before the starting of another one.

Two special cases are simulated to reveal the dynamic nature of each algorithm. Case one is simulated on a relatively sparse network with $n = 100$ and $d = 6$. After the destination oriented DAG is established, 100 random link faults are generated and all four algorithms are separately simulated. In this case, both the RBFT-based and non-RBFT-based algorithms have similar costs; while the RBFT-based algorithms usually provide shorter routing paths. Case two is simulated on a relatively dense network with $n = 100$ and $d = 18$. This time more random link faults (200 in total) are generated to cancel out the higher redundancy offered in a dense network. Because each node has more outgoing links and the average distance from the destination is 2-3 hops, the cost of non-RBFT-based algorithms is significantly lower than in the first case. The RBFT-based algorithms have a relatively higher cost, but provide significantly shorter routing paths than the non-RBFT-based algorithms. The results of a single simulation for each case is demonstrated in Figures 5-8. In addition, we repeat each special case 100 times to produce the average results shown in Table I.

First we compare the performance in terms of the average length of a routing path. For the RBFT-based algorithms (i.e., RO and RS), marked nodes can find near optimal paths to the destination by following the distance information embedded in the RBFT. For the non-RBFT-based algorithms (i.e., FR and PR), each node can choose a neighbor with the minimum watermark as its next hop to the destination. However, watermark does not provide precise distance information and, therefore, may produce longer routing paths. Here we evaluate two methods of choosing the next hop: random selection (RND) or minimum-watermark-based selection (WM). These two methods are also used by unmarked nodes in the RBFT-based algorithms. According to Table I, when next hops of unmarked nodes are randomly selected, the average routing distance of the non-RBFT-based algorithms is about 20% longer than the RBFT-based algorithms in relatively sparse networks and about 80% longer in relatively dense networks. As shown in Figure 5, the performance of the RBFT-based algorithms is best at the beginning of the simulation and deteriorates as link

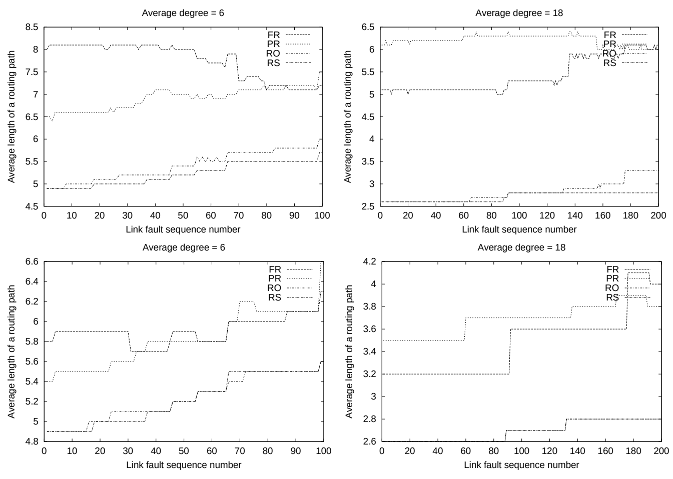

Fig. 5. Average length of a routing path in a relatively sparse network (the left column) and a relatively dense network (the right column), where each node selects its next hop randomly (the upper row) or based on the watermark (the lower row).

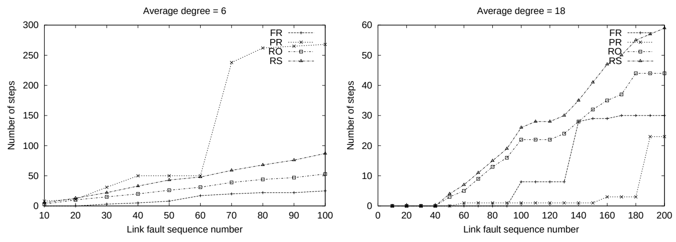

Fig. 6. Total number of steps used to handle link faults in a relatively sparse network (left) and a relatively dense network (right).

faults accumulate and the RBFT shrinks. When next hops are selected based on watermarks, the non-RBFT-based algorithms perform much better. However, their average routing distance is still about 10% longer in the relatively sparse network and about 30% longer in the relatively dense network. The difference between the two RBFT-based algorithms, RO and RS, are relatively small.

Then we compare the overheads in terms of the number of control messages and steps. A RBFT has higher maintenance cost than a normal destination oriented DAG for two reasons. Firstly, the RBFT has higher frequency of maintenance operations. In a normal destination oriented DAG, no maintenance is needed before a node loses all its outgoing links. In a RBFT, more than one maintenance operation may be needed before a node loses all its outgoing links. For example, a node that originally has several replacements may switch its parents for several times. Secondly, the RBFT has higher cost per link change. In a normal destination oriented DAG, a link reversal costs one step and one broadcast message. In a RBFT, because each node maintains a list of its children and grandchildren, a parent switch usually takes two steps and three control messages, including a broadcast message sent by the child, a message sent by the former parent to the former grandparent and another message sent by the current parent to the current grandparent. However, the RBFT can avoid

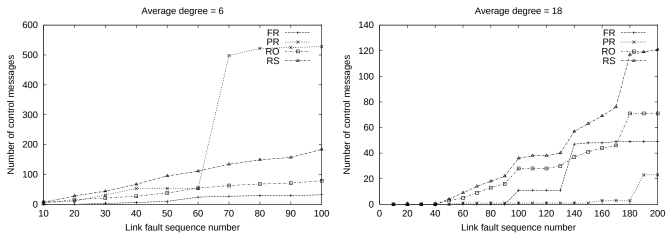

Fig. 7. Total number of control messages caused by link faults in a relatively sparse network (left) and a relatively dense network (right).

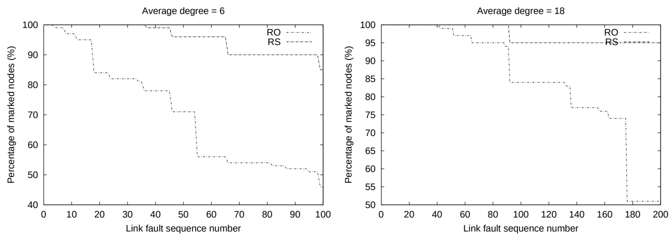

Fig. 8. Percentage of marked nodes in a relatively sparse (left) and a relatively dense (right) network.

the high cost in some cases where the DAG is dramatically changed and several links are reversed more than once. We actually observed this situation in the first case study between the 60th and the 70th link faults of the partial reversal (PR) algorithm (see Figures 6 and 7).

According to Table I, the RBFT-based algorithms have higher time costs (in steps) and message costs than the non-RBFT-based algorithms. However, the maintenance costs of the RBFT-based algorithms are still quite low and affordable (less than 2 messages per link fault), and the cost of RO is lower (less than 1.5 message per link fault). Figures 6 and 7 show that, in the relatively sparse network, maintenance operations start from the very beginning of the simulation. In the relatively dense network, there is almost no maintenance cost for the first 40 (for RO and RS) and 90 (for FR and PR) link faults. That explains why the maintenance cost is much lower than that in the sparse network. In the dense network, the chance that a faulty link is in the RBFT is lower, and is significantly lower when a faulty link is the last outgoing link of a node.

The RBFT-based algorithms also have the extra overhead of periodical global flooding. The amortized cost of rebuilding the RBFT is determined by the flooding frequency as well as the cost of each flooding. In our case studies, establishing a RBFT in the relatively sparse network takes 12 steps and 292 messages. It takes 7 steps and 282 messages in the relatively dense network. Figure 8 shows that, in the relatively sparse network, 85% (RS) and 45% (RO) nodes remain marked after the first 100 link faults. In the relatively dense network, 95% (RS) and 50% (RO) nodes remain marked after the first 200 link faults. If the rebuilding of the RBFT is triggered when less than 50% nodes are marked, the amortized cost for the replacement-only (RO) algorithm is 3 messages per link fault in the sparse network and 1.5 messages per link fault in the dense network. This cost is significantly higher than the regular maintenance cost of the RBFT. Note that this cost is still much lower than the SPT-based algorithm. Because in a RBFT-based algorithm, all unmarked nodes can still reach the destination by following the destination oriented DAG, it is acceptable even if 50% nodes are unmarked. In a SPT-based algorithm like DSDV, a node that is trimmed from the SPT loses its connection to the destination until the next global flooding. Therefore, flooding is much more frequent in those algorithms, because it is unacceptable to have 50% nodes disconnected from the destination. Similarly, the amortized cost of RS shall be significantly lower than the SPT-based algorithms, as it can maintain the RBFT much longer than RO.

6. Conclusions

We have proposed a new method of maintaining communication between nodes of a dynamic network. The idea is based on Gafni and Bertsekas' link reversal by maintaining a DAG with destination as its sink node. Our approach is based on augmenting the DAG with distance information captured in a reversed breadth-first tree. This tree itself is dynamic which may shrink (upon a delete operation) or grow (upon a network flush through flooding). The proposed approach can potentially be applied in ad hoc wireless networks. The validity of the proposed approach has been backed up through simulation. The simulation results show that the proposed approach provides shorter loop-free paths than the original GB method with lower overhead than that in the SPT-based algorithm. In our future work, we will compare various trade-offs more closely through simulation.

References

[1] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, Aug. 1980.

[2] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. *Proc. of ACM MOBICOM'99*, pages 263–270, 1999.

[3] E. M. Gafni and D. P. Bertsekas. Distributed algorithms for generating loop-free routes in neworks with frequently changing topology. *IEEE Transactions on Communications*, 29(1):11–18, 1981.

[4] D. B. Johnson. Routing in ad hoc networks of mobile hosts. *Proc. of Workshop on Mobile Computing Systems and Applications*, Dec. 1994. 158-163.

[5] P. Narvaez, K. Y. Siu, and H. Y. Tzeng. New dynamic algorithms for shortest path tree computation. *IEEE/ACM Transactions on Networking*, 8(6):734–746, Dec. 2000.

[6] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. *Proc. of IEEE INFOCOM*, pages 1405–1413, 1997.

[7] J. Raju and J. J. Garcia-Luna-Aceves. A new approach to on-demand loop-free multipath routing. *Proceedings of the International Conference on Computer Communications and Networks (IC3N)*, pages 522–527, 1999.

[8] S. Vutukury and J. J. Garcia-Luna-Aceves. MDVA: a distance-vector multipath routing protocol. *Proceedings of the IEEE INFOCOM*, pages 557–564, 2001.

[9] J. Wu. An enhanced distributed solution for generating look-free routes in dynamic networks. Technical Reports, FAU-CS-TR-01-03, Florida Atlantic University, Jan. 2001.