Internet Engineering Task Force (IETF)
Request for Comments: 7811
Category: Standards Track
ISSN: 2070-1721

G. Enyedi
A. Csaszar
Ericsson
A. Atlas
C. Bowers
Juniper Networks
A. Gopalan
University of Arizona

June 2016

An Algorithm for Computing IP/LDP Fast Reroute Using Maximally Redundant Trees (MRT-FRR)

Abstract

This document supports the solution put forth in "An Architecture for IP/LDP Fast Reroute Using Maximally Redundant Trees (MRT-FRR)" (RFC 7812) by defining the associated MRT Lowpoint algorithm that is used in the Default MRT Profile to compute both the necessary Maximally Redundant Trees with their associated next hops and the alternates to select for MRT-FRR.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

MRT Fast Reroute requires that packets can be forwarded not only on the shortest-path tree, but also on two Maximally Redundant Trees (MRTs), referred to as the MRT-Blue and the MRT-Red. A router that experiences a local failure must also have predetermined which alternate to use. This document defines how to compute these three things for use in MRT-FRR and describes the algorithm design decisions and rationale. The algorithm is based on those presented in [MRTLinear] and expanded in [EnyediThesis]. The MRT Lowpoint algorithm is required for implementation when the Default MRT Profile is implemented.

The MRT Lowpoint Algorithm defined in this document, when used for MRT Fast-Reroute as described in [RFC7812], guarantees 100% recovery for single failures when the network is 2-connected. This guaranteed coverage does not depend on the link metrics, which an operator may be using to traffic-engineer the IP network. Thus, the link metrics and general network topology are largely decoupled from the guaranteed coverage.

Just as packets routed on a hop-by-hop basis require that each router compute a shortest-path tree that is consistent, it is necessary for each router to compute the MRT-Blue next hops and MRT-Red next hops in a consistent fashion. This document defines the MRT Lowpoint algorithm to be used as a standard in the Default MRT Profile for MRT-FRR.

A router's Forwarding Information Base (FIB) will continue to contain primary next hops for the current shortest-path tree for forwarding traffic. In addition, a router's FIB will contain primary next hops for the MRT-Blue for forwarding received traffic on the MRT-Blue and primary next hops for the MRT-Red for forwarding received traffic on the MRT-Red.

What alternate next hops a Point of Local Repair (PLR) selects need not be consistent -- but loops must be prevented. To reduce congestion, it is possible for multiple alternate next hops to be selected; in the context of MRT alternates, each of those alternate next hops would be equal-cost paths.

This document defines an algorithm for selecting an appropriate MRT alternate for consideration. Other alternates, e.g., Loop-Free Alternates (LFAs) that are downstream paths, may be preferred when available. See the "Operational Considerations" section of [RFC7812] for a more detailed discussion of combining MRT alternates with those produced by other FRR technologies.

```
[E]---[D]---|           [E]<--[D]<--|                [E]-->[D]---|
 |     |    |            |     ^    |                       |    |
 |     |    |            V     |    |                       V    V
[R]   [F]  [C]          [R]   [F]  [C]               [R]   [F]  [C]
 |     |    |                  ^    ^                 ^     |    |
 |     |    |                  |    |                 |     V    |
[A]---[B]---|           [A]-->[B]---|                [A]<--[B]<--|

      (a)                     (b)                         (c)
A 2-connected graph     MRT-Blue towards R          MRT-Red towards R
```

Figure 1

The MRT Lowpoint algorithm can handle arbitrary network topologies where the whole network graph is not 2-connected, as in Figure 2, as well as the easier case where the network graph is 2-connected (Figure 1). Each MRT is a spanning tree. The pair of MRTs provide two paths from every node X to the root of the MRTs. Those paths share the minimum number of nodes and the minimum number of links. Each such shared node is a cut-vertex. Any shared links are cut-links.

```
                        [E]---[D]---|     |---[J]
                         |     |    |     |    |
                         |     |    |     |    |
                        [R]   [F]  [C]---[G]   |
                         |     |    |     |    |
                         |     |    |     |    |
                        [A]---[B]---|     |---[H]

                       (a) a graph that is not 2-connected

         [E]<--[D]<--|         [J]        [E]-->[D]---|     |---[J]
          |     ^    |          |                |    |     |    ^
          V     |    |          |                V    V     V    |
         [R]   [F]  [C]<--[G]   |         [R]   [F]  [C]<--[G]   |
                ^    ^     ^    |          ^     |    |          |
                |    |     |    V          |     V    |          |
         [A]-->[B]---|     |---[H]        [A]<--[B]<--|         [H]

          (b) MRT-Blue towards R          (c) MRT-Red towards R

                Figure 2: A Network That Is Not 2-Connected
```

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Terminology and Definitions

Please see the Terminology section of [RFC7812] for a complete list of terminology relevant to this document. The list below does not repeat terminology introduced in that RFC.

spanning tree: A tree that contains links and that connects all nodes in the network graph.

back-edge: In the context of a spanning tree computed via a depth-first search, a back-edge is a link that connects a descendant of a node x with an ancestor of x.

partial ADAG: A subset of an Almost Directed Acyclic Graph (ADAG) that doesn't yet contain all the nodes in the block. A partial ADAG is created during the MRT Lowpoint algorithm and then expanded until all nodes in the block are included and it becomes an ADAG.

DFS: Depth-First Search

DFS ancestor: A node n is a DFS ancestor of x if n is on the DFS-tree path from the DFS root to x.

DFS descendant: A node n is a DFS descendant of x if x is on the DFS-tree path from the DFS root to n.

ear: A path along nodes that are not yet included in the Generalized ADAG (GADAG) that starts at a node that is already included in the GADAG and that ends at a node that is already included in the GADAG. The starting and ending nodes may be the same node if it is a cut-vertex.

X>>Y or Y<<X: Indicates the relationship between X and Y in a partial order, such as found in a GADAG. X>>Y means that X is higher in the partial order than Y. Y<<X means that Y is lower in the partial order than X.

X>Y or Y<X: Indicates the relationship between X and Y in the total order, such as found via a topological sort. X>Y means that X is higher in the total order than Y. Y<X means that Y is lower in the total order than X.

X ?? Y: Indicates that X is unordered with respect to Y in the partial order.

UNDIRECTED: In the GADAG, each link is marked as OUTGOING, INCOMING, or both. Until the directionality of the link is determined, the link is marked as UNDIRECTED to indicate that its direction hasn't been determined.

OUTGOING: A link marked as OUTGOING has direction in the GADAG from the interface's router to the remote end.

INCOMING: A link marked as INCOMING has direction in the GADAG from the remote end to the interface's router.

4. Algorithm Key Concepts

There are five key concepts that are critical for understanding the MRT Lowpoint algorithm. The first is the idea of partially ordering the nodes in a network graph with regard to each other and to the GADAG root. The second is the idea of finding an ear of nodes and adding them in the correct direction. The third is the idea of a Lowpoint value and how it can be used to identify cut-vertices and to find a second path towards the root. The fourth is the idea that a non-2-connected graph is made up of blocks, where a block is a

2-connected cluster, a cut-link or an isolated node. The fifth is the idea of a localroot for each node; this is used to compute ADAGs in each block.

4.1. Partial Ordering for Disjoint Paths

Given any two nodes X and Y in a graph, a particular total order means that either X<Y or X>Y in that total order. An example would be a graph where the nodes are ranked based upon their unique IP loopback addresses. In a partial order, there may be some nodes for which it can't be determined whether X<<Y or X>>Y. A partial order can be captured in a directed graph, as shown in Figure 3. In a graphical representation, a link directed from X to Y indicates that X is a neighbor of Y in the network graph and X<<Y.

```
         [A]<---[R]    [E]       R << A << B << C << D << E
          |             ^        R << A << B << F << G << H << D << E
          |             |
          V             |        Unspecified Relationships:
         [B]--->[C]--->[D]             C and F
          |             ^              C and G
          |             |              C and H
          V             |
         [F]--->[G]--->[H]
```

Figure 3: Directed Graph Showing a Partial Order

To compute MRTs, the root of the MRTs is at both the very bottom and the very top of the partial ordering. This means that from any node X, one can pick nodes higher in the order until the root is reached. Similarly, from any node X, one can pick nodes lower in the order until the root is reached. For instance, in Figure 4, from G the higher nodes picked can be traced by following the directed links and are H, D, E, and R. Similarly, from G the lower nodes picked can be traced by reversing the directed links and are F, B, A, and R. A graph that represents this modified partial order is no longer a DAG; it is termed an Almost DAG (ADAG) because if the links directed to the root were removed, it would be a DAG.

```
     [A]<---[R]<---[E]      R << A << B << C << R
      |      ^      ^       R << A << B << C << D << E << R
      |      |      |       R << A << B << F << G << H << D << E << R
      V      |      |
     [B]--->[C]--->[D]      Unspecified Relationships:
      |             ^              C and F
      |             |              C and G
      V             |              C and H
     [F]--->[G]--->[H]
```

Figure 4: ADAG Showing a Partial Order with R Lowest and Highest

Most importantly, if a node Y>>X, then Y can only appear on the increasing path from X to the root and never on the decreasing path. Similarly, if a node Z<<X, then Z can only appear on the decreasing path from X to the root and never on the increasing path.

When following the increasing paths, it is possible to pick multiple higher nodes and still have the certainty that those paths will be disjoint from the decreasing paths. For example, in the previous example, node B has multiple possibilities to forward packets along an increasing path: it can either forward packets to C or F.

4.2. Finding an Ear and the Correct Direction

For simplicity, the basic idea of creating a GADAG by adding ears is described assuming that the network graph is a single 2-connected cluster so that an ADAG is sufficient. Generalizing to multiple blocks is done by considering the block-roots instead of the GADAG root -- and the actual algorithm is given in Section 5.5.

In order to understand the basic idea of finding an ADAG, first suppose that we have already a partial ADAG, which doesn't contain all the nodes in the block yet, and we want to extend it to cover all the nodes. Suppose that we find a path from a node X to Y such that X and Y are already contained by our partial ADAG, but all the remaining nodes along the path are not added to the ADAG yet. We refer to such a path as an "ear".

Recall that our ADAG is closely related to a partial order. More precisely, if we remove root R, the remaining DAG describes a partial order of the nodes. If we suppose that neither X nor Y is the root, we may be able to compare them. If one of them is definitely lesser with respect to our partial order (say X<<Y), we can add the new path to the ADAG in a direction from X to Y. As an example, consider Figure 5.

```
           E---D---|              E<--D---|           E<--D<--|
           |   |   |              |   ^   |           |   ^   |
           |   |   |              V   |   |           V   |   |
           R   F   C              R   F   C           R   F   C
           |   |   |              |   ^   |           |   ^   ^
           |   |   |              V   |   |           V   |   |
           A---B---|              A-->B---|           A-->B---|

              (a)                    (b)                 (c)

           (a) A 2-connected graph
           (b) Partial ADAG (C is not included)
           (c) Resulting ADAG after adding path (or ear) B-C-D
```

Figure 5

In this partial ADAG, node C is not yet included. However, we can find path B-C-D, where both endpoints are contained by this partial ADAG (we say those nodes are "ready" in the following text), and the remaining node (node C) is not contained yet. If we remove R, the remaining DAG defines a partial order, and with respect to this partial order, we can say that B<<D; so, we can add the path to the ADAG in the direction from B to D (arcs B->C and C->D are added). If B>>D, we would add the same path in reverse direction.

If, in the partial order where an ear's two ends are X and Y, X<<Y, then there must already be a directed path from X to Y in the ADAG. The ear must be added in a direction such that it doesn't create a cycle; therefore, the ear must go from X to Y.

In the case when X and Y are not ordered with each other, we can select either direction for the ear. We have no restriction since neither of the directions can result in a cycle. In the corner case when one of the endpoints of an ear, say X, is the root (recall that the two endpoints must be different), we could use both directions again for the ear because the root can be considered both as smaller and as greater than Y. However, we strictly pick that direction in which the root is lower than Y. The logic for this decision is explained in Section 5.7

A partial ADAG is started by finding a cycle from the root R back to itself. This can be done by selecting a non-ready neighbor N of R and then finding a path from N to R that doesn't use any links between R and N. The direction of the cycle can be assigned either way since it is starting the ordering.

Once a partial ADAG is already present, it will always have a node that is not the root R in it. The following is a brief proof that a partial ADAG can always have ears added to it: just select a non-ready neighbor N of a ready node Q, such that Q is not the root R, find a path from N to the root R in the graph with Q removed. This path is an ear where the first node of the ear is Q, the next is N, then the path until the first ready node the path reached (that ready node is the other endpoint of the path). Since the graph is 2-connected, there must be a path from N to R without Q.

It is always possible to select a non-ready neighbor N of a ready node Q so that Q is not the root R. Because the network is 2-connected, N must be connected to two different nodes and only one can be R. Because the initial cycle has already been added to the ADAG, there are ready nodes that are not R. Since the graph is 2-connected, while there are non-ready nodes, there must be a non-ready neighbor N of a ready node that is not R.

```
 Generic_Find_Ears_ADAG(root)
    Create an empty ADAG.  Add root to the ADAG.
    Mark root as IN_GADAG.
    Select an arbitrary cycle containing root.
    Add the arbitrary cycle to the ADAG.
    Mark cycle's nodes as IN_GADAG.
    Add cycle's non-root nodes to process_list.
    While there exist connected nodes in graph that are not IN_GADAG
       Select a new ear.  Let its endpoints be X and Y.
       If Y is root or (Y<<X)
          Add the ear towards X to the ADAG
       Else // (a) X is root, or (b) X<<Y, or (c) X, Y not ordered
          Add the ear towards Y to the ADAG
```

Figure 6: Generic Algorithm to Find Ears and Their Direction in 2-Connected Graph

The algorithm in Figure 6 merely requires that a cycle or ear be selected without specifying how. Regardless of the method for selecting the path, we will get an ADAG. The method used for finding and selecting the ears is important; shorter ears result in shorter paths along the MRTs. The MRT Lowpoint algorithm uses the Lowpoint Inheritance method for constructing an ADAG (and ultimately a GADAG). This method is defined in Section 5.5. Other methods for constructing GADAGs are described in Appendices B and C. An evaluation of these different methods is given in Section 8.

As an example, consider Figure 5 again. First, we select the shortest cycle containing R, which can be R-A-B-F-D-E (uniform link costs were assumed), so we get to the situation depicted in

Figure 5(b). Finally, we find a node next to a ready node; that must be node C and assume we reached it from ready node B. We search a path from C to R without B in the original graph. The first ready node along this is node D, so the open ear is B-C-D. Since B<<D, we add arc B->C and C->D to the ADAG. Since all the nodes are ready, we stop at this point.

4.3. Lowpoint Values and Their Uses

A basic way of computing a spanning tree on a network graph is to run a DFS, such as given in Figure 7. This tree has the important property that if there is a link (x, n), then either n is a DFS ancestor of x or n is a DFS descendant of x. In other words, either n is on the path from the root to x or x is on the path from the root to n.

```
global_variable: dfs_number

DFS_Visit(node x, node parent)
   D(x) = dfs_number
   dfs_number += 1
   x.dfs_parent = parent
   for each link (x, w)
     if D(w) is not set
       DFS_Visit(w, x)

Run_DFS(node gadag_root)
   dfs_number = 0
   DFS_Visit(gadag_root, NONE)
```

Figure 7: Basic DFS Algorithm

Given a node x, one can compute the minimal DFS number of the neighbors of x, i.e., min(D(w) if (x,w) is a link). This gives the earliest attachment point neighboring x. What is interesting, though, is the earliest attachment point from x and x's descendants. This is what is determined by computing the Lowpoint value.

In order to compute the low point value, the network is traversed using DFS and the vertices are numbered based on the DFS walk. Let this number be represented as DFS(x). All the edges that lead to already-visited nodes during DFS walk are back-edges. The back-edges are important because they give information about reachability of a node via another path.

The low point number is calculated by finding:

```
Low(x) = Minimum of (  (DFS(x),
   Lowest DFS(n, x->n is a back-edge),
   Lowest Low(n, x->n is tree edge in DFS walk) ).
```

A detailed algorithm for computing the lowpoint value is given in Figure 8. Figure 9 illustrates how the Lowpoint algorithm applies to an example graph.

```
global_variable: dfs_number

Lowpoint_Visit(node x, node parent, interface p_to_x)
   D(x) = dfs_number
   L(x) = D(x)
   dfs_number += 1
   x.dfs_parent = parent
   x.dfs_parent_intf = p_to_x.remote_intf
   x.lowpoint_parent = NONE
   for each ordered_interface intf of x
     if D(intf.remote_node) is not set
       Lowpoint_Visit(intf.remote_node, x, intf)
       if L(intf.remote_node) < L(x)
          L(x) = L(intf.remote_node)
          x.lowpoint_parent = intf.remote_node
          x.lowpoint_parent_intf = intf
     else if intf.remote_node is not parent
       if D(intf.remote_node) < L(x)
          L(x) = D(intf.remote_node)
          x.lowpoint_parent = intf.remote_node
          x.lowpoint_parent_intf = intf

Run_Lowpoint(node gadag_root)
   dfs_number = 0
   Lowpoint_Visit(gadag_root, NONE, NONE)
```

Figure 8: Computing Lowpoint Value

```
            [E]---|    [J]-------[I]   [P]---[O]
             |    |     |         |     |     |
             |    |     |         |     |     |
            [R]  [D]---[C]--[F]  [H]---[K]   [N]
             |          |    |    |     |     |
             |          |    |    |     |     |
            [A]--------[B]  [G]---|    [L]---[M]
```

(a) a non-2-connected graph

```
             [E]----|    [J]---------[I]    [P]------[O]
            (5, )   |  (10, )       (9, ) (16,  ) (15,  )
              |     |     |           |      |        |
              |     |     |           |      |        |
             [R]   [D]---[C]---[F]   [H]----[K]      [N]
            (0, ) (4, ) (3, ) (6, ) (8, ) (11, )  (14, )
              |           |     |     |      |        |
              |           |     |     |      |        |
             [A]---------[B]   [G]----|     [L]------[M]
            (1, )       (2, ) (7, )       (12,  )  (13,  )
```

(b) with DFS values assigned   (D(x), L(x))

```
             [E]----|    [J]---------[I]    [P]------[O]
            (5,0)   |  (10,3)       (9,3) (16,11) (15,11)
              |     |     |           |      |        |
              |     |     |           |      |        |
             [R]   [D]---[C]---[F]   [H]----[K]      [N]
            (0,0) (4,0) (3,0) (6,3) (8,3) (11,11) (14,11)
              |           |     |     |      |        |
              |           |     |     |      |        |
             [A]---------[B]   [G]----|     [L]------[M]
            (1,0)       (2,0) (7,3)       (12,11)  (13,11)
```

(c) with lowpoint values assigned (D(x), L(x))

Figure 9: Example Lowpoint Value Computation

From the lowpoint value and lowpoint parent, there are three very useful things that motivate our computation.

First, if there is a child c of x such that L(c) >= D(x), then there are no paths in the network graph that go from c or its descendants to an ancestor of x; therefore, x is a cut-vertex. In Figure 9, this can be seen by looking at the DFS children of C. C has two children, D and F and L(F) = 3 = D(C); so, it is clear that C is a cut-vertex and F is in a block where C is the block's root. L(D) = 0<3 = D(C), so D has a path to the ancestors of C; in this case, D can go via E to reach R. Comparing the lowpoint values of all a node's DFS-children with the node's DFS-value is very useful because it allows identification of the cut-vertices and thus the blocks.

Second, by repeatedly following the path given by lowpoint_parent, there is a path from x back to an ancestor of x that does not use the link [x, x.dfs_parent] in either direction. The full path need not be taken, but this gives a way of finding an initial cycle and then ears.

Third, as seen in Figure 9, even if L(x)<D(x), there may be a block that contains both the root and a DFS-child of a node while other DFS-children might be in different blocks. In this example, C's child D is in the same block as R while F is not. It is important to realize that the root of a block may also be the root of another block.

4.4. Blocks in a Graph

A key idea for the MRT Lowpoint algorithm is that any non-2-connected graph is made up by blocks (e.g., 2-connected clusters, cut-links, and/or isolated nodes). To compute GADAGs and thus MRTs, computation is done in each block to compute ADAGs or Redundant Trees and then those ADAGs or Redundant Trees are combined into a GADAG or MRT.

```
                  [E]---|    [J]-------[I]   [P]---[O]
                   |    |     |         |     |     |
                   |    |     |         |     |     |
                  [R]  [D]---[C]--[F]  [H]---[K]   [N]
                   |          |    |    |     |     |
                   |          |    |    |     |     |
                  [A]--------[B]  [G]---|    [L]---[M]

                  (a)  A graph with four blocks:
                       three 2-connected clusters
                       and one cut-link


                  [E]<--|    [J]<------[I]   [P]<--[O]
                   |    |     |         ^     |     ^
                   V    |     V         |     V     |
                  [R]  [D]<--[C]  [F]  [H]<---[K]  [N]
                              ^    |    ^           ^
                              |    V    |           |
                  [A]------->[B]  [G]---|     [L]-->[M]

                    (b) MRT-Blue for destination R


                  [E]---|    [J]-------->[I]    [P]-->[O]
                        |                 |            |
                        V                 V            V
                  [R]  [D]-->[C]<---[F]  [H]<---[K]   [N]
                   ^          |      ^    |      ^     |
                   |          V      |    |      |     V
                  [A]<-------[B]    [G]<--|     [L]<--[M]

                     (c) MRT-Red for destination R


                                 Figure 10
```

Consider the example depicted in Figure 10 (a). In this figure, a special graph is presented, showing us all the ways 2-connected clusters can be connected. It has four blocks: block 1 contains R, A, B, C, D, E; block 2 contains C, F, G, H, I, J; block 3 contains K, L, M, N, O, P; and block 4 is a cut-link containing H and K. As can be observed, the first two blocks have one common node (node C) and blocks 2 and 3 do not have any common node, but they are connected through a cut-link that is block 4. No two blocks can have more than one common node, since two blocks with at least two common nodes would qualify as a single 2-connected cluster.

Moreover, observe that if we want to get from one block to another, we must use a cut-vertex (the cut-vertices in this graph are C, H, K), regardless of the path selected, so we can say that all the paths from block 3 along the MRTs rooted at R will cross K first. This observation means that if we want to find a pair of MRTs rooted at R, then we need to build up a pair of RTs in block 3 with K as a root. Similarly, we need to find another pair of RTs in block 2 with C as a root, and finally, we need the last pair of RTs in block 1 with R as a root. When all the trees are selected, we can simply combine them; when a block is a cut-link (as in block 4), that cut-link is added in the same direction to both of the trees. The resulting trees are depicted in Figure 10 (b) and (c).

Similarly, to create a GADAG it is sufficient to compute ADAGs in each block and connect them.

It is necessary, therefore, to identify the cut-vertices, the blocks and identify the appropriate localroot to use for each block.

4.5. Determining Localroot and Assigning Block-ID

Each node in a network graph has a localroot, which is the cut-vertex (or root) in the same block that is closest to the root. The localroot is used to determine whether two nodes share a common block.

```
Compute_Localroot(node x, node localroot)
    x.localroot = localroot
    for each DFS child node c of x
        if L(c) < D(x)   //x is not a cut-vertex
            Compute_Localroot(c, x.localroot)
        else
            mark x as cut-vertex
            Compute_Localroot(c, x)

Compute_Localroot(gadag_root, gadag_root)
```

Figure 11: A Method for Computing Localroots

There are two different ways of computing the localroot for each node. The stand-alone method is given in Figure 11 and better illustrates the concept; it is used by the GADAG construction methods given in Appendices B and C. The MRT Lowpoint algorithm computes the localroot for a block as part of computing the GADAG using lowpoint inheritance; the essence of this computation is given in Figure 12. Both methods for computing the localroot produce the same results.

```
Get the current node, s.
Compute an ear (either through lowpoint inheritance
or by following dfs parents) from s to a ready node e.
(Thus, s is not e, if there is such ear.)
if s is e
   for each node x in the ear that is not s
       x.localroot = s
else
   for each node x in the ear that is not s or e
       x.localroot = e.localroot
```

Figure 12: Ear-Based Method for Computing Localroots

Once the localroots are known, two nodes X and Y are in a common block if and only if one of the following three conditions apply.

o Y's localroot is X's localroot : They are in the same block and neither is the cut-vertex closest to the root.

o Y's localroot is X: X is the cut-vertex closest to the root for Y's block

o Y is X's localroot: Y is the cut-vertex closest to the root for X's block

Once we have computed the localroot for each node in the network graph, we can assign for each node, a Block-ID that represents the block in which the node is present. This computation is shown in Figure 13.

```
global_var: max_block_id

Assign_Block_ID(x, cur_block_id)
  x.block_id = cur_block_id
  foreach DFS child c of x
     if (c.local_root is x)
        max_block_id += 1
        Assign_Block_ID(c, max_block_id)
     else
       Assign_Block_ID(c, cur_block_id)

max_block_id = 0
Assign_Block_ID(gadag_root, max_block_id)
```

Figure 13: Assigning Block-ID to Identify Blocks

5. MRT Lowpoint Algorithm Specification

The MRT Lowpoint algorithm computes one GADAG that is then used by a router to determine its MRT-Blue and MRT-Red next hops to all destinations. Finally, based upon that information, alternates are selected for each next hop to each destination. The different parts of this algorithm are described below.

- o Order the interfaces in the network graph. See Section 5.1.
- o Compute the local MRT Island for the particular MRT Profile. See Section 5.2.
- o Select the root to use for the GADAG. See Section 5.3.
- o Initialize all interfaces to UNDIRECTED. See Section 5.4.
- o Compute the DFS value, e.g., D(x), and lowpoint value, L(x). See Figure 8.
- o Construct the GADAG. See Section 5.5.
- o Assign directions to all interfaces that are still UNDIRECTED. See Section 5.6.
- o From the computing router x, compute the next hops for the MRT-Blue and MRT-Red. See Section 5.7.
- o Identify alternates for each next hop to each destination by determining which one of the MRT-Blue and the MRT-Red the computing router x should select. See Section 5.8.

A Python implementation of this algorithm is given in Appendix A.

5.1. Interface Ordering

To ensure consistency in computation, all routers MUST order interfaces identically down to the set of links with the same metric to the same neighboring node. This is necessary for the DFS in Lowpoint_Visit in Section 4.3, where the selection order of the interfaces to explore results in different trees. Consistent interface ordering is also necessary for computing the GADAG, where the selection order of the interfaces to use to form ears can result in different GADAGs. It is also necessary for the topological sort described in Section 5.8, where different topological sort orderings can result in undirected links being added to the GADAG in different directions.

The required ordering between two interfaces from the same router x is given in Figure 14.

```
   Interface_Compare(interface a, interface b)
     if a.metric < b.metric
        return A_LESS_THAN_B
     if b.metric < a.metric
        return B_LESS_THAN_A
     if a.neighbor.mrt_node_id < b.neighbor.mrt_node_id
        return A_LESS_THAN_B
     if b.neighbor.mrt_node_id < a.neighbor.mrt_node_id
        return B_LESS_THAN_A
     // Same metric to same node, so the order doesn't matter for
     // interoperability.
     return A_EQUAL_TO_B
```

Figure 14: Rules for Ranking Multiple Interfaces (Order Is from Low to High)

In Figure 14, if two interfaces on a router connect to the same remote router with the same metric, the Interface_Compare function returns A_EQUAL_TO_B. This is because the order in which those interfaces are initially explored does not affect the final GADAG produced by the algorithm described here. While only one of the links will be added to the GADAG in the initial traversal, the other parallel links will be added to the GADAG with the same direction assigned during the procedure for assigning direction to UNDIRECTED links described in Section 5.6. An implementation is free to apply some additional criteria to break ties in interface ordering in this situation, but those criteria are not specified here since they will not affect the final GADAG produced by the algorithm.

The Interface_Compare function in Figure 14 relies on the interface.metric and the interface.neighbor.mrt_node_id values to order interfaces. The exact source of these values for different IGPs and applications is specified in Figure 15. The metric and mrt_node_id values for OSPFv2, OSPFv3, and IS-IS provided here is normative. The metric and mrt_node_id values for IS-IS Path Control and Reservation (PCR) in this table should be considered informational. The normative values are specified in [IEEE8021Qca].

IGP/flooding protocol and application	mrt_node_id of neighbor on interface	metric of interface
OSPFv2 for IP/LDP FRR	4-octet Neighbor Router ID in Link ID field for corresponding point-to-point link in Router-LSA	2-octet Metric field for corresponding point-to-point link in Router-LSA
OSPFv3 for IP/LDP FRR	4-octet Neighbor Router ID field for corresponding point-to-point link in Router-LSA	2-octet Metric field for corresponding point-to-point link in Router-LSA
IS-IS for IP/LDP FRR	7-octet neighbor system ID and pseudonode number in Extended IS Reachability TLV (type 22) or Multi-Topology IS Neighbor TLV (type 222)	3-octet metric field in Extended IS Reachability TLV (type 22) or Multi-Topology IS Neighbor TLV (type 222)
IS-IS PCR for protection of traffic in bridged networks	8-octet Bridge ID created from 2-octet Bridge Priority in Shortest Path Bridging SPB Instance sub-TLV (type 1) carried in MT-Capability TLV (type 144) and 6-octet neighbor system ID in Extended IS Reachability TLV (type 22) or Multi-Topology Intermediate Systems TLV (type 222) (informational)	3-octet SPB-LINK-METRIC in SPB-Metric sub-TLV (type 29) in Extended IS Reachability TLV (type 22) or Multi-Topology Intermediate Systems TLV (type 222). In the case of asymmetric link metrics, the larger link metric is used for both link directions. (informational)

Figure 15: Value of interface.neighbor.mrt_node_id and interface.metric to Be Used for Ranking Interfaces, for Different Flooding Protocols and Applications

The metrics are unsigned integers and MUST be compared as unsigned integers. The results of mrt_node_id comparisons MUST be the same as would be obtained by converting the mrt_node_ids to unsigned integers using network byte order and performing the comparison as unsigned integers. In the case of IS-IS for IP/LDP FRR with point-to-point links, the pseudonode number (the 7th octet) is zero. Broadcast interfaces will be discussed in Section 7.

5.2. MRT Island Identification

The local MRT Island for a particular MRT profile can be determined by starting from the computing router in the network graph and doing a breadth-first-search (BFS). The BFS explores only links that are in the same area/level, are not IGP-excluded, and are not MRT-ineligible. The BFS explores only nodes that support the particular MRT profile. See Section 7 of [RFC7812] for more-precise definitions of these criteria.

```
MRT_Island_Identification(topology, computing_rtr, profile_id, area)
  for all routers in topology
      rtr.IN_MRT_ISLAND = FALSE
  computing_rtr.IN_MRT_ISLAND = TRUE
  explore_list = { computing_rtr }
  while (explore_list is not empty)
     next_rtr = remove_head(explore_list)
     for each intf in next_rtr
        if (not intf.IN_MRT_ISLAND
           and not intf.MRT-ineligible
           and not intf.remote_intf.MRT-ineligible
           and not intf.IGP-excluded and (intf in area)
           and (intf.remote_node supports profile_id) )
           intf.IN_MRT_ISLAND = TRUE
           intf.remote_intf.IN_MRT_ISLAND = TRUE
           if (not intf.remote_node.IN_MRT_ISLAND))
              intf.remote_node.IN_MRT_ISLAND = TRUE
              add_to_tail(explore_list, intf.remote_node)
```

Figure 16: MRT Island Identification

5.3. GADAG Root Selection

In Section 8.3 of [RFC7812], the GADAG Root Selection Policy is described for the Default MRT Profile. This selection policy allows routers to consistently select a common GADAG Root inside the local MRT Island, based on advertised priority values. The MRT Lowpoint algorithm simply requires that all routers in the MRT Island MUST select the same GADAG Root; the mechanism can vary based upon the MRT profile description. Before beginning computation, the network graph is reduced to contain only the set of routers that support the specific MRT profile whose MRTs are being computed.

As noted in Section 7, pseudonodes MUST NOT be considered for GADAG root selection.

It is expected that an operator will designate a set of routers as good choices for selection as GADAG root by setting the GADAG Root Selection Priority for that set of routers to lower (more preferred) numerical values. For guidance on setting the GADAG Root Selection Priority values, refer to Section 9.1.

5.4. Initialization

Before running the algorithm, there is the standard type of initialization to be done, such as clearing any computed DFS-values, lowpoint-values, DFS parents, lowpoint-parents, any MRT-computed next hops, and flags associated with algorithm.

It is assumed that a regular SPF computation has been run so that the primary next hops from the computing router to each destination are known. This is required for determining alternates at the last step.

Initially, all interfaces MUST be initialized to UNDIRECTED. Whether they are OUTGOING, INCOMING, or both is determined when the GADAG is constructed and augmented.

It is possible that some links and nodes will be marked using standard IGP mechanisms to discourage or prevent transit traffic. Section 7.3.1 of [RFC7812] describes how those links and nodes are excluded from MRT Island formation.

MRT-FRR also has the ability to advertise links MRT-Ineligible, as described in Section 7.3.2 of [RFC7812]. These links are excluded from the MRT Island and the GADAG. Computation of MRT next hops will therefore not use any MRT-ineligible links. The MRT Lowpoint algorithm does still need to consider MRT-ineligible links when computing FRR alternates, because an MRT-ineligible link can still be the shortest-path next hop to reach a destination.

When a broadcast interface is advertised as MRT-ineligible, then the pseudonode representing the entire broadcast network MUST NOT be included in the MRT Island. This is equivalent to excluding all of the broadcast interfaces on that broadcast network from the MRT Island.

5.5. Constructing the GADAG Using Lowpoint Inheritance

As discussed in Section 4.2, it is necessary to find ears from a node x that is already in the GADAG (known as IN_GADAG). Two different methods are used to find ears in the algorithm. The first is by going to a DFS-child that is not IN_GADAG and then following the chain of lowpoint parents until an IN_GADAG node is found. The second is by going to a neighbor that is not IN_GADAG and then following the chain of DFS parents until an IN_GADAG node is found. As an ear is found, the associated interfaces are marked based on the direction taken. The nodes in the ear are marked as IN_GADAG. In the algorithm, first the ears via DFS-children are found and then the ears via DFS-neighbors are found.

By adding both types of ears when an IN_GADAG node is processed, all ears that connect to that node are found. The order in which the IN_GADAG nodes are processed is, of course, key to the algorithm. The order is a stack of ears so the most recent ear is found at the top of the stack. Of course, the stack stores nodes and not ears, so an ordered list of nodes, from the first node in the ear to the last node in the ear, is created as the ear is explored and then that list is pushed onto the stack.

Each ear represents a partial order (see Figure 4) and processing the nodes in order along each ear ensures that all ears connecting to a node are found before a node higher in the partial order has its ears explored. This means that the direction of the links in the ear is always from the node x being processed towards the other end of the ear. Additionally, by using a stack of ears, this means that any unprocessed nodes in previous ears can only be ordered higher than nodes in the ears below it on the stack.

In this algorithm that depends upon Lowpoint inheritance, it is necessary that every node has a lowpoint parent that is not itself. If a node is a cut-vertex, that may not yet be the case. Therefore, any nodes without a lowpoint parent will have their lowpoint parent set to their DFS parent and their lowpoint value set to the DFS-value of their parent. This assignment also properly allows an ear between two cut-vertices.

Finally, the algorithm simultaneously computes each node's localroot, as described in Figure 12. This is further elaborated as follows. The localroot can be inherited from the node at the end of the ear unless the end of the ear is x itself, in which case the localroot for all the nodes in the ear would be x. This is because whenever the first cycle is found in a block, or an ear involving a bridge is computed, the cut-vertex closest to the root would be x itself. In all other scenarios, the properties of lowpoint/dfs parents ensure that the end of the ear will be in the same block, and thus inheriting its localroot would be the correct localroot for all newly added nodes.

The pseudocode for the GADAG algorithm (assuming that the adjustment of lowpoint for cut-vertices has been made) is shown in Figure 17.

```
Construct_Ear(x, Stack, intf, ear_type)
   ear_list = empty
   cur_node = intf.remote_node
   cur_intf = intf
   not_done = true

   while not_done
      cur_intf.UNDIRECTED = false
      cur_intf.OUTGOING = true
      cur_intf.remote_intf.UNDIRECTED = false
      cur_intf.remote_intf.INCOMING = true

      if cur_node.IN_GADAG is false
         cur_node.IN_GADAG = true
         add_to_list_end(ear_list, cur_node)
         if ear_type is CHILD
            cur_intf = cur_node.lowpoint_parent_intf
            cur_node = cur_node.lowpoint_parent
         else  // ear_type must be NEIGHBOR
            cur_intf = cur_node.dfs_parent_intf
            cur_node = cur_node.dfs_parent
      else
         not_done = false

   if (ear_type is CHILD) and (cur_node is x)
      // x is a cut-vertex and the local root for
      // the block in which the ear is computed
      x.IS_CUT_VERTEX = true
      localroot = x
   else
      // Inherit localroot from the end of the ear
      localroot = cur_node.localroot
   while ear_list is not empty
      y = remove_end_item_from_list(ear_list)
      y.localroot = localroot
      push(Stack, y)

Construct_GADAG_via_Lowpoint(topology, gadag_root)
  gadag_root.IN_GADAG = true
  gadag_root.localroot = None
  Initialize Stack to empty
```

```
          push gadag_root onto Stack
          while (Stack is not empty)
             x = pop(Stack)
             foreach ordered_interface intf of x
                if ((intf.remote_node.IN_GADAG == false) and
                    (intf.remote_node.dfs_parent is x))
                    Construct_Ear(x, Stack, intf, CHILD)
             foreach ordered_interface intf of x
                if ((intf.remote_node.IN_GADAG == false) and
                    (intf.remote_node.dfs_parent is not x))
                    Construct_Ear(x, Stack, intf, NEIGHBOR)

         Construct_GADAG_via_Lowpoint(topology, gadag_root)
```

Figure 17: Lowpoint Inheritance GADAG Algorithm

5.6. Augmenting the GADAG by Directing All Links

The GADAG, regardless of the method used to construct it, at this point could be used to find MRTs, but the topology does not include all links in the network graph. That has two impacts. First, there might be shorter paths that respect the GADAG partial ordering and so the alternate paths would not be as short as possible. Second, there may be additional paths between a router x and the root that are not included in the GADAG. Including those provides potentially more bandwidth to traffic flowing on the alternates and may reduce congestion compared to just using the GADAG as currently constructed.

The goal is thus to assign direction to every remaining link marked as UNDIRECTED to improve the paths and number of paths found when the MRTs are computed.

To do this, we need to establish a total order that respects the partial order described by the GADAG. This can be done using Kahn's topological sort [Kahn_1962_topo_sort], which essentially assigns a number to a node x only after all nodes before it (e.g., with a link incoming to x) have had their numbers assigned. The only issue with the topological sort is that it works on DAGs and not ADAGs or GADAGs.

To convert a GADAG to a DAG, it is necessary to remove all links that point to a root of block from within that block. That provides the necessary conversion to a DAG and then a topological sort can be done. When adding undirected links to the GADAG, links connecting the block root to other nodes in that block need special handling because the topological order will not always give the right answer for those links. There are three cases to consider. If the undirected link in question has another parallel link between the same two nodes that is already directed, then the direction of the undirected link can be inherited from the previously directed link. In the case of parallel cut links, we set all of the parallel links to both INCOMING and OUTGOING. Otherwise, the undirected link in question is set to OUTGOING from the block root node. A cut-link can then be identified by the fact that it will be directed both INCOMING and OUTGOING in the GADAG. The exact details of this whole process are captured in Figure 18.

```
Add_Undirected_Block_Root_Links(topo, gadag_root)
    foreach node x in topo
        if x.IS_CUT_VERTEX or x is gadag_root
            foreach interface i of x
                if (i.remote_node.localroot is not x
                                  or i.PROCESSED )
                    continue
                Initialize bundle_list to empty
                bundle.UNDIRECTED = true
                bundle.OUTGOING = false
                bundle.INCOMING = false
                foreach interface i2 in x
                    if i2.remote_node is i.remote_node
                        add_to_list_end(bundle_list, i2)
                        if not i2.UNDIRECTED:
                            bundle.UNDIRECTED = false
                            if i2.INCOMING:
                                bundle.INCOMING = true
                            if i2.OUTGOING:
                                bundle.OUTGOING = true
                if bundle.UNDIRECTED
                    foreach interface i3 in bundle_list
                        i3.UNDIRECTED = false
                        i3.remote_intf.UNDIRECTED = false
                        i3.PROCESSED = true
                        i3.remote_intf.PROCESSED = true
                        i3.OUTGOING = true
                        i3.remote_intf.INCOMING = true
                else
                    if (bundle.OUTGOING and bundle.INCOMING)
                        foreach interface i3 in bundle_list
                            i3.UNDIRECTED = false
                            i3.remote_intf.UNDIRECTED = false
                            i3.PROCESSED = true
                            i3.remote_intf.PROCESSED = true
                            i3.OUTGOING = true
                            i3.INCOMING = true
                            i3.remote_intf.INCOMING = true
                            i3.remote_intf.OUTGOING = true
```

```
                         else if bundle.OUTGOING
                             foreach interface i3 in bundle_list
                                 i3.UNDIRECTED = false
                                 i3.remote_intf.UNDIRECTED = false
                                 i3.PROCESSED = true
                                 i3.remote_intf.PROCESSED = true
                                 i3.OUTGOING = true
                                 i3.remote_intf.INCOMING = true
                         else if bundle.INCOMING
                             foreach interface i3 in bundle_list
                                 i3.UNDIRECTED = false
                                 i3.remote_intf.UNDIRECTED = false
                                 i3.PROCESSED = true
                                 i3.remote_intf.PROCESSED = true
                                 i3.INCOMING = true
                                 i3.remote_intf.OUTGOING = true

     Modify_Block_Root_Incoming_Links(topo, gadag_root)
         foreach node x in topo
             if x.IS_CUT_VERTEX or x is gadag_root
                 foreach interface i of x
                     if i.remote_node.localroot is x
                         if i.INCOMING:
                             i.INCOMING = false
                             i.INCOMING_STORED = true
                             i.remote_intf.OUTGOING = false
                             i.remote_intf.OUTGOING_STORED = true

     Revert_Block_Root_Incoming_Links(topo, gadag_root)
         foreach node x in topo
             if x.IS_CUT_VERTEX or x is gadag_root
                 foreach interface i of x
                     if i.remote_node.localroot is x
                         if i.INCOMING_STORED
                             i.INCOMING = true
                             i.remote_intf.OUTGOING = true
                             i.INCOMING_STORED = false
                             i.remote_intf.OUTGOING_STORED = false

     Run_Topological_Sort_GADAG(topo, gadag_root)
         Modify_Block_Root_Incoming_Links(topo, gadag_root)
         foreach node x in topo
             node.unvisited = 0
             foreach interface i of x
                 if (i.INCOMING)
                     node.unvisited += 1
         Initialize working_list to empty
         Initialize topo_order_list to empty
         add_to_list_end(working_list, gadag_root)
         while working_list is not empty
             y = remove_start_item_from_list(working_list)
             add_to_list_end(topo_order_list, y)
             foreach ordered_interface i of y
                 if intf.OUTGOING
                     i.remote_node.unvisited -= 1
                     if i.remote_node.unvisited is 0
                         add_to_list_end(working_list, i.remote_node)
         next_topo_order = 1
         while topo_order_list is not empty
             y = remove_start_item_from_list(topo_order_list)
             y.topo_order = next_topo_order
             next_topo_order += 1
         Revert_Block_Root_Incoming_Links(topo, gadag_root)

     def Set_Other_Undirected_Links_Based_On_Topo_Order(topo)
         foreach node x in topo
             foreach interface i of x
                 if i.UNDIRECTED:
                     if x.topo_order < i.remote_node.topo_order
                         i.OUTGOING = true
                         i.UNDIRECTED = false
                         i.remote_intf.INCOMING = true
                         i.remote_intf.UNDIRECTED = false
                     else
                         i.INCOMING = true
                         i.UNDIRECTED = false
                         i.remote_intf.OUTGOING = true
                         i.remote_intf.UNDIRECTED = false

     Add_Undirected_Links(topo, gadag_root)
         Add_Undirected_Block_Root_Links(topo, gadag_root)
         Run_Topological_Sort_GADAG(topo, gadag_root)
         Set_Other_Undirected_Links_Based_On_Topo_Order(topo)

     Add_Undirected_Links(topo, gadag_root)
```

Figure 18: Assigning Direction to UNDIRECTED Links

Proxy-nodes do not need to be added to the network graph. They cannot be transited and do not affect the MRTs that are computed. The details of how the MRT-Blue and MRT-Red next hops are computed for proxy-nodes and how the appropriate alternate next hops are selected is given in Section 5.9.

5.7. Compute MRT Next Hops

As was discussed in Section 4.1, once an ADAG is found, it is straightforward to find the next hops from any node X to the ADAG root. However, in this algorithm, we will reuse the common GADAG and find not only the one pair of MRTs rooted at the GADAG root with it, but find a pair rooted at each node. This is useful since it is significantly faster to compute.

The method for computing differently rooted MRTs from the common GADAG is based on two ideas. First, if two nodes X and Y are ordered with respect to each other in the partial order, then an SPF along OUTGOING links (an increasing-SPF) and an SPF along INCOMING links (a decreasing-SPF) can be used to find the increasing and decreasing paths. Second, if two nodes X and Y aren't ordered with respect to each other in the partial order, then intermediary nodes can be used to create the paths by increasing/decreasing to the intermediary and then decreasing/increasing to reach Y.

As usual, the two basic ideas will be discussed assuming the network is 2-connected. The generalization to multiple blocks is discussed in Section 5.7.4. The full algorithm is given in Section 5.7.5.

5.7.1. MRT Next Hops to All Nodes Ordered with Respect to the Computing Node

Finding two node-disjoint paths from the computing router X to any node Y depends upon whether Y>>X or Y<<X. As shown in Figure 19, if Y>>X, then there is an increasing path that goes from X to Y without crossing R; this contains nodes in the interval [X,Y]. There is also a decreasing path that decreases towards R and then decreases from R to Y; this contains nodes in the interval [X,R-small] or [R-great,Y]. The two paths cannot have common nodes other than X and Y.

```
                     [Y]<---(Cloud 2)<--- [X]
                      |                    ^
                      |                    |
                      V                    |
                   (Cloud 3)--->[R]--->(Cloud 1)

                  MRT-Blue path: X->Cloud 2->Y
                  MRT-Red path: X->Cloud 1->R->Cloud 3->Y
```

Figure 19: Y>>X

Similar logic applies if Y<<X, as shown in Figure 20. In this case, the increasing path from X increases to R and then increases from R to Y to use nodes in the intervals [X,R-great] and [R-small, Y]. The decreasing path from X reaches Y without crossing R and uses nodes in the interval [Y,X].

```
                 [X]<---(Cloud 2)<--- [Y]
                  |                    ^
                  |                    |
                  V                    |
                (Cloud 3)--->[R]--->(Cloud 1)

               MRT-Blue path: X->Cloud 3->R->Cloud 1->Y
               MRT-Red path: X->Cloud 2->Y
```

Figure 20: Y<<X

5.7.2. MRT Next Hops to All Nodes Not Ordered with Respect to the Computing Node

When X and Y are not ordered, the first path should increase until we get to a node G, where G>>Y. At G, we need to decrease to Y. The other path should be just the opposite: we must decrease until we get to a node H, where H<<Y, and then increase. Since R is smaller and greater than Y, such G and H must exist. It is also easy to see that these two paths must be node disjoint: the first path contains nodes in interval [X,G] and [Y,G], while the second path contains nodes in interval [H,X] and [H,Y]. This is illustrated in Figure 21. It is necessary to decrease and then increase for the MRT-Blue and increase and then decrease for the MRT-Red; if one simply increased for one and decreased for the other, then both paths would go through the root R.

```
                 (Cloud 6)<---[Y]<---(Cloud 5)<------------|
                   |                                       |
                   |                                       |
                   V                                       |
                  [G]--->(Cloud 4)--->[R]--->(Cloud 1)--->[H]
                   ^                                       |
                   |                                       |
                   |                                       |
                  (Cloud 3)<---[X]<---(Cloud 2)<-----------|

              MRT-Blue path: decrease to H and increase to Y
                   X->Cloud 2->H->Cloud 5->Y
              MRT-Red path:  increase to G and decrease to Y
                   X->Cloud 3->G->Cloud 6->Y
```

Figure 21: X and Y Unordered

This gives disjoint paths as long as G and H are not the same node. Since G>>Y and H<<Y, if G and H could be the same node, that would have to be the root R. This is not possible because there is only one incoming interface to the root R that is created when the initial cycle is found. Recall from Figure 6 that whenever an ear was found to have an end that was the root R, the ear was directed from R so that the associated interface on R is outgoing and not incoming. Therefore, there must be exactly one node M that is the largest one before R, so the MRT-Red path will never reach R; it will turn at M and decrease to Y.

5.7.3. Computing Redundant Tree Next Hops in a 2-Connected Graph

The basic ideas for computing RT next hops in a 2-connected graph were given in Sections 5.7.1 and 5.7.2. Given these two ideas, how can we find the trees?

If some node X only wants to find the next hops (which is usually the case for IP networks), it is enough to find which nodes are greater and less than X, and which are not ordered; this can be done by running an increasing-SPF and a decreasing-SPF rooted at X and not exploring any links from the ADAG root.

In principle, a traversal method other than SPF could be used to traverse the GADAG in the process of determining blue and red next hops that result in maximally redundant trees. This will be the case as long as one traversal uses the links in the direction specified by the GADAG and the other traversal uses the links in the direction opposite of that specified by the GADAG. However, a different traversal algorithm will generally result in different blue and red next hops. Therefore, the algorithm specified here requires the use of SPF to traverse the GADAG to generate MRT blue and red next hops, as described below.

An increasing-SPF rooted at X and not exploring links from the root will find the increasing next hops to all Y>>X. Those increasing next hops are X's next hops on the MRT-Blue to reach Y. A decreasing-SPF rooted at X and not exploring links from the root will find the decreasing next hops to all Z<<X. Those decreasing next hops are X's next hops on the MRT-Red to reach Z. Since the root R is both greater than and less than X, after this increasing-SPF and decreasing-SPF, X's next hops on the MRT-Blue and on the MRT-Red to reach R are known. For every node Y>>X, X's next hops on the MRT-Red to reach Y are set to those on the MRT-Red to reach R. For every node Z<<X, X's next hops on the MRT-Blue to reach Z are set to those on the MRT-Blue to reach R.

For those nodes that were not reached by either the increasing-SPF or the decreasing-SPF, we can determine the next hops as well. The increasing MRT-Blue next hop for a node that is not ordered with respect to X is the next hop along the decreasing MRT-Red towards R, and the decreasing MRT-Red next hop is the next hop along the increasing MRT-Blue towards R. Naturally, since R is ordered with respect to all the nodes, there will always be an increasing and a decreasing path towards it. This algorithm does not provide the complete specific path taken but just the appropriate next hops to use. The identities of G and H are not determined by the computing node X.

The final case to consider is when the GADAG root R computes its own next hops. Since the GADAG root R is << all other nodes, running an increasing-SPF rooted at R will reach all other nodes; the MRT-Blue next hops are those found with this increasing-SPF. Similarly, since the GADAG root R is >> all other nodes, running a decreasing-SPF rooted at R will reach all other nodes; the MRT-Red next hops are those found with this decreasing-SPF.

```
                 E---D---|              E<--D<--|
                 |   |   |              |   ^   |
                 |   |   |              V   |   |
                 R   F   C              R   F   C
                 |   |   |              |   ^   ^
                 |   |   |              V   |   |
                 A---B---|              A-->B---|

                    (a)                    (b)
            A 2-connected graph    A spanning ADAG rooted at R
```

Figure 22

As an example, consider the situation depicted in Figure 22. Node C runs an increasing-SPF and a decreasing-SPF on the ADAG. The increasing-SPF reaches D, E, and R; the decreasing-SPF reaches B, A, and R. E>>C. So, towards E the MRT-Blue next hop is D, since E was reached on the increasing path through D. The MRT-Red next hop towards E is B, since R was reached on the decreasing path through B. Since E>>D, D will similarly compute its MRT-Blue next hop to be E, ensuring that a packet on MRT-Blue will use path C-D-E. B, A, and R will similarly compute the MRT-Red next hops towards E (which is ordered less than B, A and R), ensuring that a packet on MRT-Red will use path C-B-A-R-E.

C can determine the next hops towards F as well. Since F is not ordered with respect to C, the MRT-Blue next hop is the decreasing one towards R (which is B) and the MRT-Red next hop is the increasing one towards R (which is D). Since F>>B, for its MRT-Blue next hop towards F, B will use the real increasing next hop towards F. So a packet forwarded to B on MRT-Blue will get to F on path C-B-F. Similarly, D will use the real decreasing next hop towards F as its MRT-Red next hop, a packet on MRT-Red will use path C-D-F.

5.7.4. Generalizing for a Graph That Isn't 2-Connected

If a graph isn't 2-connected, then the basic approach given in Section 5.7.3 needs some extensions to determine the appropriate MRT next hops to use for destinations outside the computing router X's blocks. In order to find a pair of maximally redundant trees in that graph, we need to find a pair of RTs in each of the blocks (the root of these trees will be discussed later) and combine them.

When computing the MRT next hops from a router X, there are three basic differences:

1. Only nodes in a common block with X should be explored in the increasing-SPF and decreasing-SPF.

2. Instead of using the GADAG root, X's localroot should be used. This has the following implications:

   A. The links from X's localroot should not be explored.

   B. If a node is explored in the outgoing SPF so Y>>X, then X's MRT-Red next hops to reach Y uses X's MRT-Red next hops to reach X's localroot and if Z<<X, then X's MRT-Blue next hops to reach Z uses X's MRT-Blue next hops to reach X's localroot.

   C. If a node W in a common block with X was not reached in the increasing-SPF or decreasing-SPF, then W is unordered with respect to X. X's MRT-Blue next hops to W are X's decreasing (aka MRT-Red) next hops to X's localroot. X's MRT-Red next hops to W are X's increasing (aka MRT-Blue) next hops to X's localroot.

3. For nodes in different blocks, the next hops must be inherited via the relevant cut-vertex.

These are all captured in the detailed algorithm given in Section 5.7.5.

5.7.5. Complete Algorithm to Compute MRT Next Hops

The complete algorithm to compute MRT Next Hops for a particular router X is given in Figure 23. In addition to computing the MRT-Blue next hops and MRT-Red next hops used by X to reach each node Y, the algorithm also stores an "order_proxy", which is the proper cut-vertex to reach Y if it is outside the block, and which is used later in deciding whether the MRT-Blue or the MRT-Red can provide an acceptable alternate for a particular primary next hop.

```
   In_Common_Block(x, y)
     if ( (x.block_id is y.block_id)
          or (x is y.localroot) or (y is x.localroot) )
        return true
     return false

   Store_Results(y, direction)
      if direction is FORWARD
         y.higher = true
         y.blue_next_hops = y.next_hops
      if direction is REVERSE
         y.lower = true
         y.red_next_hops = y.next_hops

   SPF_No_Traverse_Block_Root(spf_root, block_root, direction)
      Initialize spf_heap to empty
      Initialize nodes' spf_metric to infinity and next_hops to empty
      spf_root.spf_metric = 0
      insert(spf_heap, spf_root)
      while (spf_heap is not empty)
          min_node = remove_lowest(spf_heap)
          Store_Results(min_node, direction)
          if ((min_node is spf_root) or (min_node is not block_root))
             foreach interface intf of min_node
                if ( ( ((direction is FORWARD) and intf.OUTGOING) or
                     ((direction is REVERSE) and intf.INCOMING) )
                     and In_Common_Block(spf_root, intf.remote_node) )
                   path_metric = min_node.spf_metric + intf.metric
                   if path_metric < intf.remote_node.spf_metric
                      intf.remote_node.spf_metric = path_metric
                      if min_node is spf_root
                        intf.remote_node.next_hops = make_list(intf)
                      else
                        intf.remote_node.next_hops = min_node.next_hops
                      insert_or_update(spf_heap, intf.remote_node)
                   else if path_metric == intf.remote_node.spf_metric
                      if min_node is spf_root
                         add_to_list(intf.remote_node.next_hops, intf)
                      else
                         add_list_to_list(intf.remote_node.next_hops,
                                          min_node.next_hops)

   SetEdge(y)
     if y.blue_next_hops is empty and y.red_next_hops is empty
        SetEdge(y.localroot)
        y.blue_next_hops = y.localroot.blue_next_hops
        y.red_next_hops = y.localroot.red_next_hops
        y.order_proxy = y.localroot.order_proxy
```

```
Compute_MRT_NextHops(x, gadag_root)
   foreach node y
     y.higher = y.lower = false
     clear y.red_next_hops and y.blue_next_hops
     y.order_proxy = y
   SPF_No_Traverse_Block_Root(x, x.localroot, FORWARD)
   SPF_No_Traverse_Block_Root(x, x.localroot, REVERSE)

   // red and blue next hops are stored to x.localroot as different
   // paths are found via the SPF and reverse-SPF.
   // Similarly, any node whose localroot is x will have its
   // red_next_hops and blue_next_hops already set.

   // Handle nodes in the same block that aren't the localroot
   foreach node y
     if (y.IN_MRT_ISLAND and (y is not x) and
          (y.block_id is x.block_id) )
        if y.higher
           y.red_next_hops = x.localroot.red_next_hops
        else if y.lower
           y.blue_next_hops = x.localroot.blue_next_hops
        else
           y.blue_next_hops = x.localroot.red_next_hops
           y.red_next_hops = x.localroot.blue_next_hops

   // Inherit next hops and order_proxies to other components
   if (x is not gadag_root) and (x.localroot is not gadag_root)
      gadag_root.blue_next_hops = x.localroot.blue_next_hops
      gadag_root.red_next_hops = x.localroot.red_next_hops
      gadag_root.order_proxy = x.localroot
   foreach node y
      if (y is not gadag_root) and (y is not x) and y.IN_MRT_ISLAND
        SetEdge(y)

max_block_id = 0
Assign_Block_ID(gadag_root, max_block_id)
Compute_MRT_NextHops(x, gadag_root)
```

Figure 23: Complete Algorithm to Compute MRT Next Hops

5.8. Identify MRT Alternates

At this point, a computing router S knows its MRT-Blue next hops and MRT-Red next hops for each destination in the MRT Island. The primary next hops along the SPT are also known. It remains to determine for each primary next hop to a destination D, which MRT avoids the primary next-hop node F. This computation depends upon data set in Compute_MRT_NextHops such as each node y's y.blue_next_hops, y.red_next_hops, y.order_proxy, y.higher, y.lower, and topo_orders. Recall that any router knows only which are the nodes greater and lesser than itself, but it cannot decide the relation between any two given nodes easily; that is why we need topological ordering.

For each primary next-hop node F to each destination D, S can call Select_Alternates(S, D, F, primary_intf) to determine whether to use the MRT-Blue or MRT-Red next hops as the alternate next hop(s) for that primary next hop. The algorithm is given in Figure 24 and discussed afterwards.

```
Select_Alternates_Internal(D, F, primary_intf,
                              D_lower, D_higher, D_topo_order):
    if D_higher and D_lower
        if F.HIGHER and F.LOWER
            if F.topo_order < D_topo_order
                return USE_RED
            else
                return USE_BLUE
        if F.HIGHER
            return USE_RED
        if F.LOWER
            return USE_BLUE
        //F unordered wrt S
        return USE_RED_OR_BLUE

    else if D_higher
        if F.HIGHER and F.LOWER
            return USE_BLUE
        if F.LOWER
            return USE_BLUE
        if F.HIGHER
            if (F.topo_order > D_topo_order)
                return USE_BLUE
            if (F.topo_order < D_topo_order)
                return USE_RED
        //F unordered wrt S
        return USE_RED_OR_BLUE

    else if D_lower
        if F.HIGHER and F.LOWER
            return USE_RED
        if F.HIGHER
            return USE_RED
        if F.LOWER
            if F.topo_order > D_topo_order
                return USE_BLUE
            if F.topo_order < D_topo_order
                return USE_RED
        //F unordered wrt S
        return USE_RED_OR_BLUE

    else  //D is unordered wrt S
        if F.HIGHER and F.LOWER
            if primary_intf.OUTGOING and primary_intf.INCOMING
                return USE_RED_OR_BLUE
            if primary_intf.OUTGOING
                return USE_BLUE
            if primary_intf.INCOMING
                return USE_RED
            //primary_intf not in GADAG
            return USE_RED
        if F.LOWER
            return USE_RED
        if F.HIGHER
            return USE_BLUE
        //F unordered wrt S
        if F.topo_order > D_topo_order:
            return USE_BLUE
        else:
            return USE_RED

Select_Alternates(D, F, primary_intf)
    if not In_Common_Block(F, S)
        return PRIM_NH_IN_DIFFERENT_BLOCK
    if (D is F) or (D.order_proxy is F)
        return PRIM_NH_IS_D_OR_OP_FOR_D
    D_lower = D.order_proxy.LOWER
    D_higher = D.order_proxy.HIGHER
    D_topo_order = D.order_proxy.topo_order
    return Select_Alternates_Internal(D, F, primary_intf,
                                  D_lower, D_higher, D_topo_order)
```

Figure 24: Select_Alternates() and Select_Alternates_Internal()

It is useful to first handle the case where F is also D, or F is the order proxy for D. In this case, only link protection is possible. The MRT that doesn't use the failed primary next hop is used. If both MRTs use the primary next hop, then the primary next hop must be a cut-link, so either MRT could be used but the set of MRT next hops must be pruned to avoid the failed primary next-hop interface. To indicate this case, Select_Alternates returns PRIM_NH_IS_D_OR_OP_FOR_D. Explicit pseudocode to handle the three sub-cases above is not provided.

The logic behind Select_Alternates_Internal() is described in Figure 25. As an example, consider the first case described in the table, where the D>>S and D<<S. If this is true, then either S or D must be the block root, R. If F>>S and F<<S, then S is the block root. So the blue path from S to D is the increasing path to D, and the red path S to D is the decreasing path to D. If the F.topo_order>D.topo_order, then either F is ordered higher than D or F is unordered with respect to D. Therefore, F is either on a decreasing path from S to D, or it is on neither an increasing nor a decreasing path from S to D. In either case, it is safe to take an increasing path from S to D to avoid F. We know that when S is R, the increasing path is the blue path, so it is safe to use the blue path to avoid F.

If instead F.topo_order<D.topo_order, then either F is ordered lower than D, or F is unordered with respect to D. Therefore, F is either on an increasing path from S to D, or it is on neither an increasing nor a decreasing path from S to D. In either case, it is safe to take a decreasing path from S to D to avoid F. We know that when S is R, the decreasing path is the red path, so it is safe to use the red path to avoid F.

If F>>S or F<<S (but not both), then D is the block root. We then know that the blue path from S to D is the increasing path to R, and the red path is the decreasing path to R. When F>>S, we deduce that F is on an increasing path from S to R. So in order to avoid F, we use a decreasing path from S to R, which is the red path. Instead, when F<<S, we deduce that F is on a decreasing path from S to R. So in order to avoid F, we use an increasing path from S to R, which is the blue path.

All possible cases are systematically described in the same manner in the rest of the table.

D wrt S	MRT blue and red path properties	F wrt S	additional criteria	F wrt MRT (deduced)	Alternate
D>>S and D<<S, D is R,	Blue path: Increasing path to R. Red path: Decreasing path to R.	F>>S only	additional criteria not needed	F on an increasing path from S to R	Use Red to avoid F
or		F<<S only	additional criteria not needed	F on a decreasing path from S to R	Use Blue to avoid F
S is R	Blue path: Increasing path to D. Red path: Decreasing path to D.	F>>S and F<<S,	topo(F)>topo(D) implies that F>>D or F??D	F on a decreasing path from S to D or neither	Use Blue to avoid F
			topo(F)<topo(D) implies that F<<D or F??D	F on an increasing path from S to D or neither	Use Red to avoid F
		F??S	Can only occur when link between F and S is marked MRT_INELIGIBLE	F is on neither increasing nor decr. path from S to D or R	Use Red or Blue to avoid F

D>>S only	Blue path: Increasing shortest path from S to D. Red path: Decreasing shortest path from S to R, then decreasing shortest path from R to D.	F<<S only	additional criteria not needed	F on decreasing path from S to R	Use Blue to avoid F
		F>>S only	topo(F)>topo(D) implies that F>>D or F??D	F on decreasing path from R to D or neither	Use Blue to avoid F
			topo(F)<topo(D) implies that F<<D or F??D	F on increasing path from S to D or neither	Use Red to avoid F
		F>>S and F<<S, F is R	additional criteria not needed	F on Red	Use Blue to avoid F
		F??S	Can only occur when link between F and S is marked MRT_INELIGIBLE	F is on neither increasing nor decr. path from S to D or R	Use Red or Blue to avoid F

D<<S only	Blue path: Increasing shortest path from S to R, then increasing shortest path from R to D. Red path: Decreasing shortest path from S to D.	F>>S only	additional criteria not needed	F on increasing path from S to R	Use Red to avoid F
		F<<S only	topo(F)>topo(D) implies that F>>D or F??D	F on decreasing path from R to D or neither	Use Blue to avoid F
			topo(F)<topo(D) implies that F<<D or F??D	F on increasing path from S to D or neither	Use Red to avoid F
		F>>S and F<<S, F is R	additional criteria not needed	F on Blue	Use Red to avoid F
		F??S	Can only occur when link between F and S is marked MRT_INELIGIBLE	F is on neither increasing nor decr. path from S to D or R	Use Red or Blue to avoid F
D??S	Blue path: Decr. from S to first node K<<D, then incr. to D. Red path: Incr. from S to first node L>>D, then decr.	F<<S only	additional criteria not needed	F on a decreasing path from S to K.	Use Red to avoid F
		F>>S only	additional criteria not needed	F on an increasing path from S to L	Use Blue to avoid F

		F??S	topo(F)>topo(D) implies that F>>D or F??D	F on decr. path from L to D or neither	Use Blue to avoid F
			topo(F)<topo(D) implies that F<<D or F??D	F on incr. path from K to D or neither	Use Red to avoid F
		F>>S and F<<S, F is R	GADAG link direction S->F	F on an incr. path from S	Use Blue to avoid F
			GADAG link direction S<-F	F on a decr. path from S	Use Red to avoid F
			GADAG link direction S<-->F	Either F is the order proxy for D (case already handled) or D is in a different block from F, in which case Red or Blue avoids F	
			S-F link not in GADAG, only when S-F link is MRT_INELIGIBLE	Relies on special construction of GADAG to demonstrate that using Red avoids F (see text)	

Determining MRT next hops and alternates based on the partial order and topological sort relationships between the source(S), destination(D), primary next hop(F), and block root(R). topo(N) indicates the topological sort value of node N. X??Y indicates that node X is unordered with respect to node Y. It is assumed that the case where F is D, or where F is the order proxy for D, has already been handled.

Figure 25: Determining MRT Next Hops and Alternates

The last case in Figure 25 requires additional explanation. The fact that the red path from S to D in this case avoids F relies on a special property of the GADAGs that we have constructed in this algorithm, a property not shared by all GADAGs in general. When D is unordered with respect to S, and F is the localroot for S, it can occur that the link between S and F is not in the GADAG only when that link has been marked MRT_INELIGIBLE. For an arbitrary GADAG, S doesn't have enough information based on the computed order relationships to determine if the red path or blue path will hit F (which is also the localroot) before hitting K or L, and making it safely to D. However, the GADAGs that we construct using the algorithm in this document are not arbitrary GADAGs. They have the additional property that incoming links to a localroot come from only one other node in the same block. This is a result of the method of construction. This additional property guarantees that the red path from S to D will never pass through the localroot of S. (That would require the localroot to play the role of L, the first node in the path ordered higher than D, which would in turn require the localroot to have two incoming links in the GADAG, which cannot happen.) Therefore, it is safe to use the red path to avoid F with these specially constructed GADAGs.

As an example of how Select_Alternates_Internal() operates, consider the ADAG depicted in Figure 26 and first suppose that G is the source, D is the destination, and H is the failed next hop. Since D>>G, we need to compare H.topo_order and D.topo_order. Since D.topo_order>H.topo_order, D must be either higher than H or unordered with respect to H, so we should select the decreasing path towards the root. If, however, the destination were instead J, we must find that H.topo_order>J.topo_order, so we must choose the increasing Blue next hop to J, which is I. In the case, when instead the destination is C, we find that we need to first decrease to avoid using H, so the Blue, first decreasing then increasing, path is selected.

```
[E]<-[D]<-[H]<-[J]
 |    ^    ^    ^
 V    |    |    |
[R]  [C]  [G]->[I]
 |    ^    ^    ^
 V    |    |    |
[A]->[B]->[F]---|
```

Figure 26: ADAG Rooted at R for a 2-Connected Graph

5.9. Named Proxy-Nodes

As discussed in Section 11.2 of [RFC7812], it is necessary to find MRT-Blue and MRT-Red next hops and MRT-FRR alternates for named proxy-nodes. An example use case is for a router that is not part of that local MRT Island, when there is only partial MRT support in the domain.

5.9.1. Determining Proxy-Node Attachment Routers

Section 11.2 of [RFC7812] discusses general considerations for determining the two proxy-node attachment routers for a given proxy-node, corresponding to a prefix. A router in the MRT Island that advertises the prefix is a candidate for being a proxy-node attachment router, with the associated named-proxy-cost equal to the advertised cost to the prefix.

An Island Border Router (IBR) is a router in the MRT Island that is connected to an Island Neighbor (IN), which is a router not in the MRT Island but in the same area/level. An (IBR,IN) pair is a candidate for being a proxy-node attachment router, if the shortest path from the IN to the prefix does not enter the MRT Island. A method for identifying such Loop-Free Island Neighbors (LFINs) is given below. The named-proxy-cost assigned to each (IBR, IN) pair is cost(IBR, IN) + D_opt(IN, prefix).

From the set of prefix-advertising routers and the set of IBRs with at least one LFIN, the two routers with the lowest named-proxy-cost are selected. Ties are broken based upon the lowest Router ID. For ease of discussion, the two selected routers will be referred to as proxy-node attachment routers.

5.9.2. Computing If an Island Neighbor (IN) Is Loop-Free

As discussed above, the IN needs to be loop-free with respect to the whole MRT Island for the destination. This can be accomplished by running the usual SPF algorithm while keeping track of which shortest paths have passed through the MRT island. Pseudocode for this is shown in Figure 27. The Island_Marking_SPF() is run for each IN that needs to be evaluated for the loop-free condition, with the IN as the spf_root. Whether or not an IN is loop-free with respect to the MRT island can then be determined by evaluating node.PATH_HITS_ISLAND for each destination of interest.

```
 Island_Marking_SPF(spf_root)
     Initialize spf_heap to empty
     Initialize nodes' spf_metric to infinity and next_hops to empty
         and PATH_HITS_ISLAND to false
     spf_root.spf_metric = 0
     insert(spf_heap, spf_root)
     while (spf_heap is not empty)
         min_node = remove_lowest(spf_heap)
         foreach interface intf of min_node
             path_metric = min_node.spf_metric + intf.metric
             if path_metric < intf.remote_node.spf_metric
                 intf.remote_node.spf_metric = path_metric
                 if min_node is spf_root
                     intf.remote_node.next_hops = make_list(intf)
                 else
                     intf.remote_node.next_hops = min_node.next_hops
                 if intf.remote_node.IN_MRT_ISLAND
                     intf.remote_node.PATH_HITS_ISLAND = true
                 else
                     intf.remote_node.PATH_HITS_ISLAND =
                         min_node.PATH_HITS_ISLAND
                 insert_or_update(spf_heap, intf.remote_node)
             else if path_metric == intf.remote_node.spf_metric
                 if min_node is spf_root
                     add_to_list(intf.remote_node.next_hops, intf)
                 else
                     add_list_to_list(intf.remote_node.next_hops,
                                      min_node.next_hops)
                 if intf.remote_node.IN_MRT_ISLAND
                     intf.remote_node.PATH_HITS_ISLAND = true
                 else
                     intf.remote_node.PATH_HITS_ISLAND =
                         min_node.PATH_HITS_ISLAND
```

Figure 27: Island_Marking_SPF() for Determining If an Island Neighbor Is Loop-Free

It is also possible that a given prefix is originated by a combination of non-island routers and island routers. The results of the Island_Marking_SPF() computation can be used to determine if the shortest path from an IN to reach that prefix hits the MRT Island. The shortest path for the IN to reach prefix P is determined by the total cost to reach prefix P, which is the sum of the cost for the IN to reach a prefix-advertising node and the cost with which that node advertises the prefix. The path with the minimum total cost to prefix P is chosen. If the prefix-advertising node for that minimum total cost path has PATH_HITS_ISLAND set to True, then the IN is not loop-free with respect to the MRT Island for reaching prefix P. If there are multiple minimum total cost paths to reach prefix P, then all of the prefix-advertising routers involved in the minimum total cost paths MUST have PATH_HITS_ISLAND set to False for the IN to be considered loop-free to reach P.

Note that there are other computations that could be used to determine if paths from a given IN _might_ pass through the MRT Island for a given prefix or destination. For example, a previous draft version of this document specified running the SPF algorithm on modified topology that treats the MRT Island as a single node (with intra-island links set to zero cost) in order to provide input to computations to determine if the path from IN to non-island destination hits the MRT Island in this modified topology. This computation is enough to guarantee that a path will not hit the MRT Island in the original topology. However, it is possible that a path that is disqualified for hitting the MRT Island in the modified topology will not actually hit the MRT Island in the original topology. The algorithm described in Island_Marking_SPF() above does not modify the original topology, and will only disqualify a path if the actual path does in fact hit the MRT Island.

Since all routers need to come to the same conclusion about which routers qualify as LFINs, this specification requires that all routers computing LFINs MUST use an algorithm whose result is identical to that of the Island_Marking_SPF() in Figure 27.

5.9.3. Computing MRT Next Hops for Proxy-Nodes

Determining the MRT next hops for a proxy-node in the degenerate case where the proxy-node is attached to only one node in the GADAG is trivial, as all needed information can be derived from that proxy-node attachment router. If there are multiple interfaces connecting the proxy-node to the single proxy-node attachment router, then some can be assigned to MRT-Red and others to MRT_Blue.

Now, consider the proxy-node P that is attached to two proxy-node attachment routers. The pseudocode for Select_Proxy_Node_NHs(P,S) in Figure 28 specifies how a computing-router S MUST compute the MRT red and blue next hops to reach proxy-node P. The proxy-node attachment router with the lower value of mrt_node_id (as defined in Figure 15) is assigned to X, and the other proxy-node attachment router is assigned to Y. We will be using the relative order of X,Y, and S in the partial order defined by the GADAG to determine the MRT red and blue next hops to reach P, so we also define A and B as the order proxies for X and Y, respectively, with respect to S. The order proxies for all nodes with respect to S were already computed in Compute_MRT_NextHops().

```
 def Select_Proxy_Node_NHs(P,S):
     if P.pnar1.node.node_id < P.pnar2.node.node_id:
         X = P.pnar1.node
         Y = P.pnar2.node
     else:
         X = P.pnar2.node
         Y = P.pnar1.node
     P.pnar_X = X
     P.pnar_Y = Y
     A = X.order_proxy
     B = Y.order_proxy
     if (A is S.localroot
         and B is S.localroot):
         // case 1.0
         Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
         Copy_List_Items(P.red_next_hops, Y.red_next_hops)
         return
     if (A is S.localroot
         and B is not S.localroot):
         // case 2.0
         if B.LOWER:
             // case 2.1
             Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
             Copy_List_Items(P.red_next_hops, Y.red_next_hops)
             return
         if B.HIGHER:
             // case 2.2
             Copy_List_Items(P.blue_next_hops, X.red_next_hops)
             Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
             return
         else:
             // case 2.3
             Copy_List_Items(P.blue_next_hops, X.red_next_hops)
             Copy_List_Items(P.red_next_hops, Y.red_next_hops)
             return
     if (A is not S.localroot
         and B is S.localroot):
         // case 3.0
         if A.LOWER:
             // case 3.1
             Copy_List_Items(P.blue_next_hops, X.red_next_hops)
             Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
             return
         if A.HIGHER:
             // case 3.2
             Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
             Copy_List_Items(P.red_next_hops, Y.red_next_hops)
             return
         else:
             // case 3.3
             Copy_List_Items(P.blue_next_hops, X.red_next_hops)
             Copy_List_Items(P.red_next_hops, Y.red_next_hops)
             return
     if (A is not S.localroot
         and B is not S.localroot):
         // case 4.0
         if (S is A.localroot or S is B.localroot):
             // case 4.05
             if A.topo_order < B.topo_order:
                 // case 4.05.1
                 Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                 Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                 return
             else:
                 // case 4.05.2
                 Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                 Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                 return
         if A.LOWER:
             // case 4.1
             if B.HIGHER:
                 // case 4.1.1
                 Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                 Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                 return
             if B.LOWER:
                 // case 4.1.2
                 if A.topo_order < B.topo_order:
                     // case 4.1.2.1
                     Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                     Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                     return
                 else:
                     // case 4.1.2.2
                     Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                     Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                     return
             else:
                 // case 4.1.3
                 Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                 Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                 return
         if A.HIGHER:
             // case 4.2
```

```
            if B.HIGHER:
                // case 4.2.1
                if A.topo_order < B.topo_order:
                    // case 4.2.1.1
                    Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                    Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                    return
                else:
                    // case 4.2.1.2
                    Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                    Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                    return
            if B.LOWER:
                // case 4.2.2
                Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                return
            else:
                // case 4.2.3
                Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                return
        else:
            // case 4.3
            if B.LOWER:
                // case 4.3.1
                Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                return
            if B.HIGHER:
                // case 4.3.2
                Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                return
            else:
                // case 4.3.3
                if A.topo_order < B.topo_order:
                    // case 4.3.3.1
                    Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                    Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                    return
```

```
                 else:
                     // case 4.3.3.2
                     Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                     Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                     return
     assert(False)
```

Figure 28: Select_Proxy_Node_NHs()

It is useful to understand up front that the blue next hops to reach proxy-node P produced by Select_Proxy_Node_NHs() will always be the next hops that reach proxy-node attachment router X, while the red next hops to reach proxy-node P will always be the next hops that reach proxy-node attachment router Y. This is different from the red and blue next hops produced by Compute_MRT_NextHops() where, for example, blue next hops to a destination that is ordered with respect to the source will always correspond to an INCREASING next hop on the GADAG. The exact choice of which next hops chosen by Select_Proxy_Node_NHs() as the blue next hops to reach P (which will necessarily go through X on its way to P) does depend on the GADAG, but the relationship is more complex than was the case with Compute_MRT_NextHops().

There are 21 different relative order relationships between A, B, and S that Select_Proxy_Node_NHs() uses to determine red and blue next hops to P. This document does not attempt to provide an exhaustive description of each case considered in Select_Proxy_Node_NHs(). Instead, we provide a high-level overview of the different cases, and we consider a few cases in detail to give an example of the reasoning that can be used to understand each case.

At the highest level, Select_Proxy_Node_NHs() distinguishes between four different cases depending on whether or not A or B is the localroot for S. For example, for case 4.0, neither A nor B is the localroot for S. Case 4.05 addresses the case where S is the localroot for either A or B, while cases 4.1, 4.2, and 4.3 address the cases where A is ordered lower than S, A is ordered higher than S, or A is unordered with respect to S on the GADAG. In general, each of these cases is then further subdivided into whether or not B is ordered lower than S, B is ordered higher than S, or B is unordered with respect to S. In some cases, we also need a further level of discrimination, where we use the topological sort order of A with respect to B.

As a detailed example, let's consider case 4.1 and all of its sub-cases, and explain why the red and blue next hops to reach P are chosen as they are in Select_Proxy_Node_NHs(). In case 4.1, neither A nor B is the localroot for S, S is not the localroot for A or B, and A is ordered lower than S on the GADAG. In this situation, we know that the red path to reach X (as computed in Compute_MRT_NextHops()) will follow DECREASING next hops towards A, while the blue path to reach X will follow INCREASING next hops to the localroot, and then INCREASING next hops to A.

Now consider sub-case 4.1.1 where B is ordered higher than S. In this situation, we know that the blue path to reach Y will follow INCREASING next hops towards B, while the red next hops to reach Y will follow DECREASING next hops to the localroot, and then DECREASING next hops to B. So, to reach X and Y by two disjoint paths, we can choose the red next hops to X and the blue next hops to Y. We have chosen the convention that blue next hops to P are those that pass through X, and red next hops to P are those that pass through Y, so we can see that case 4.1.1 produces the desired result. Choosing blue to X and red to Y does not produce disjoint paths because the paths intersect at least at the localroot.

Now consider sub-case 4.1.2 where B is ordered lower than S. In this situation, we know that the red path to reach Y will follow DECREASING next hops towards B, while the BLUE next hops to reach Y will follow INCREASING next hops to the localroot, and then INCREASING next hops to A. The choice here is more difficult than in 4.1.1 because A and B are both on the DECREASING path from S towards the localroot. We want to use the direct DECREASING(red) path to the one that is nearer to S on the GADAG. We get this extra information by comparing the topological sort order of A and B. If A.topo_order<B.topo_order, then we use red to Y and blue to X, since the red path to Y will DECREASE to B without hitting A, and the blue path to X will INCREASE to A without hitting B. Instead, if A.topo_order>B.topo_order, then we use red to X and blue to Y.

Note that when A is unordered with respect to B, the result of comparing A.topo_order with B.topo_order could be greater than or less than. In this case, the result doesn't matter because either choice (red to Y and blue to X or red to X and blue to Y) would work. What is required is that all nodes in the network give the same result when comparing A.topo_order with B.topo_order. This is guaranteed by having all nodes run the same algorithm (Run_Topological_Sort_GADAG()) to compute the topological sort order.

Finally, we consider case 4.1.3, where B is unordered with respect to S. In this case, the blue path to reach Y will follow the DECREASING next hops towards the localroot until it reaches some node (K) which is ordered less than B, after which it will take INCREASING next hops to B. The red path to reach Y will follow the INCREASING next hops towards the localroot until it reaches some node (L) which is ordered greater than B, after which it will take DECREASING next hops to B.

Both K and A are reached by DECREASING from S, but we don't have information about whether or not that DECREASING path will hit K or A first. Instead, we do know that the INCREASING path from S will hit L before reaching A. Therefore, we use the red path to reach Y and the red path to reach X.

Similar reasoning can be applied to understand the other 17 cases used in Select_Proxy_Node_NHs(). However, cases 2.3 and 3.3 deserve special attention because the correctness of the solution for these two cases relies on a special property of the GADAGs that we have constructed in this algorithm, a property not shared by all GADAGs in general. Focusing on case 2.3, we consider the case where A is the localroot for S, while B is not, and B is unordered with respect to S. The red path to X DECREASES from S to the localroot A, while the blue path to X INCREASES from S to the localroot A. The blue path to Y DECREASES towards the localroot A until it reaches some node (K) which is ordered less than B, after which the path INCREASES to B. The red path to Y INCREASES towards the localroot A until it reaches some node (L) which is ordered greater than B, after which the path DECREASES to B. It can be shown that for an arbitrary GADAG, with only the ordering relationships computed so far, we don't have enough information to choose a pair of paths to reach X and Y that are guaranteed to be disjoint. In some topologies, A will play the role of K, the first node ordered less than B on the blue path to Y. In other topologies, A will play the role of L, the first node ordered greater than B on the red path to Y. The basic problem is that we cannot distinguish between these two cases based on the ordering relationships.

As discussed Section 5.8, the GADAGs that we construct using the algorithm in this document are not arbitrary GADAGs. They have the additional property that incoming links to a localroot come from only one other node in the same block. This is a result of the method of construction. This additional property guarantees that localroot A will never play the role of L in the red path to Y, since L must have at least two incoming links from different nodes in the same block in the GADAG. This, in turn, allows Select_Proxy_Node_NHs() to choose the red path to Y and the red path to X as the disjoint MRT paths to reach P.

5.9.4. Computing MRT Alternates for Proxy-Nodes

After finding the red and the blue next hops for a given proxy-node P, it is necessary to know which one of these to use in the case of failure. This can be done by Select_Alternates_Proxy_Node(), as shown in the pseudocode in Figure 29.

```
   def Select_Alternates_Proxy_Node(P,F,primary_intf):
       S = primary_intf.local_node
       X = P.pnar_X
       Y = P.pnar_Y
       A = X.order_proxy
       B = Y.order_proxy
       if F is A and F is B:
           return 'PRIM_NH_IS_OP_FOR_BOTH_X_AND_Y'
       if F is A:
           return 'USE_RED'
       if F is B:
           return 'USE_BLUE'

       if not In_Common_Block(A, B):
           if In_Common_Block(F, A):
               return 'USE_RED'
           elif In_Common_Block(F, B):
               return 'USE_BLUE'
           else:
               return 'USE_RED_OR_BLUE'
       if (not In_Common_Block(F, A)
           and not In_Common_Block(F, A) ):
           return 'USE_RED_OR_BLUE'

       alt_to_X = Select_Alternates(X, F, primary_intf)
       alt_to_Y = Select_Alternates(Y, F, primary_intf)

       if (alt_to_X == 'USE_RED_OR_BLUE'
           and alt_to_Y == 'USE_RED_OR_BLUE'):
           return 'USE_RED_OR_BLUE'
       if alt_to_X == 'USE_RED_OR_BLUE':
           return 'USE_BLUE'
       if alt_to_Y == 'USE_RED_OR_BLUE':
           return 'USE_RED'

       if (A is S.localroot
           and B is S.localroot):
           // case 1.0
           if (alt_to_X == 'USE_BLUE' and alt_to_Y == 'USE_RED'):
               return 'USE_RED_OR_BLUE'
           if alt_to_X == 'USE_BLUE':
               return 'USE_BLUE'
           if alt_to_Y == 'USE_RED':
               return 'USE_RED'
           assert(False)
       if (A is S.localroot
           and B is not S.localroot):
           // case 2.0
```

```
          if B.LOWER:
              // case 2.1
              if (alt_to_X == 'USE_BLUE' and alt_to_Y == 'USE_RED'):
                  return 'USE_RED_OR_BLUE'
              if alt_to_X == 'USE_BLUE':
                  return 'USE_BLUE'
              if alt_to_Y == 'USE_RED':
                  return 'USE_RED'
              assert(False)
          if B.HIGHER:
              // case 2.2
              if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_BLUE'):
                  return 'USE_RED_OR_BLUE'
              if alt_to_X == 'USE_RED':
                  return 'USE_BLUE'
              if alt_to_Y == 'USE_BLUE':
                  return 'USE_RED'
              assert(False)
          else:
              // case 2.3
              if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_RED'):
                  return 'USE_RED_OR_BLUE'
              if alt_to_X == 'USE_RED':
                  return 'USE_BLUE'
              if alt_to_Y == 'USE_RED':
                  return 'USE_RED'
              assert(False)
      if (A is not S.localroot
          and B is S.localroot):
          // case 3.0
          if A.LOWER:
              // case 3.1
              if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_BLUE'):
                  return 'USE_RED_OR_BLUE'
              if alt_to_X == 'USE_RED':
                  return 'USE_BLUE'
              if alt_to_Y == 'USE_BLUE':
                  return 'USE_RED'
              assert(False)
          if A.HIGHER:
              // case 3.2
              if (alt_to_X == 'USE_BLUE' and alt_to_Y == 'USE_RED'):
                  return 'USE_RED_OR_BLUE'
              if alt_to_X == 'USE_BLUE':
                  return 'USE_BLUE'
              if alt_to_Y == 'USE_RED':
                  return 'USE_RED'
              assert(False)
          else:
              // case 3.3
              if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_RED'):
                  return 'USE_RED_OR_BLUE'
              if alt_to_X == 'USE_RED':
                  return 'USE_BLUE'
              if alt_to_Y == 'USE_RED':
                  return 'USE_RED'
              assert(False)
      if (A is not S.localroot
          and B is not S.localroot):
          // case 4.0
          if (S is A.localroot or S is B.localroot):
              // case 4.05
              if A.topo_order < B.topo_order:
                  // case 4.05.1
                  if (alt_to_X == 'USE_BLUE' and alt_to_Y == 'USE_RED'):
                      return 'USE_RED_OR_BLUE'
                  if alt_to_X == 'USE_BLUE':
                      return 'USE_BLUE'
                  if alt_to_Y == 'USE_RED':
                      return 'USE_RED'
                  assert(False)
              else:
                  // case 4.05.2
                  if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_BLUE'):
                      return 'USE_RED_OR_BLUE'
                  if alt_to_X == 'USE_RED':
                      return 'USE_BLUE'
                  if alt_to_Y == 'USE_BLUE':
                      return 'USE_RED'
                  assert(False)
          if A.LOWER:
              // case 4.1
              if B.HIGHER:
                  // case 4.1.1
                  if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_BLUE'):
                      return 'USE_RED_OR_BLUE'
                  if alt_to_X == 'USE_RED':
                      return 'USE_BLUE'
                  if alt_to_Y == 'USE_BLUE':
                      return 'USE_RED'
                  assert(False)
              if B.LOWER:
                  // case 4.1.2
                  if A.topo_order < B.topo_order:
                      // case 4.1.2.1
                      if (alt_to_X == 'USE_BLUE'
                          and alt_to_Y == 'USE_RED'):
                          return 'USE_RED_OR_BLUE'
                      if alt_to_X == 'USE_BLUE':
                          return 'USE_BLUE'
                      if alt_to_Y == 'USE_RED':
                          return 'USE_RED'
                      assert(False)
                  else:
                      // case 4.1.2.2
                      if (alt_to_X == 'USE_RED'
                          and alt_to_Y == 'USE_BLUE'):
                          return 'USE_RED_OR_BLUE'
                      if alt_to_X == 'USE_RED':
                          return 'USE_BLUE'
                      if alt_to_Y == 'USE_BLUE':
                          return 'USE_RED'
                      assert(False)
              else:
                  // case 4.1.3
                  if (F.LOWER and not F.HIGHER
                      and F.topo_order > A.topo_order):
                      // case 4.1.3.1
                      return 'USE_RED'
                  else:
                      // case 4.1.3.2
                      return 'USE_BLUE'
          if A.HIGHER:
              // case 4.2
              if B.HIGHER:
                  // case 4.2.1
                  if A.topo_order < B.topo_order:
                      // case 4.2.1.1
                      if (alt_to_X == 'USE_BLUE'
                          and alt_to_Y == 'USE_RED'):
                          return 'USE_RED_OR_BLUE'
                      if alt_to_X == 'USE_BLUE':
                          return 'USE_BLUE'
                      if alt_to_Y == 'USE_RED':
                          return 'USE_RED'
                      assert(False)
                  else:
                      // case 4.2.1.2
                      if (alt_to_X == 'USE_RED'
                          and alt_to_Y == 'USE_BLUE'):
                          return 'USE_RED_OR_BLUE'
                      if alt_to_X == 'USE_RED':
                          return 'USE_BLUE'
                      if alt_to_Y == 'USE_BLUE':
                          return 'USE_RED'
                      assert(False)
              if B.LOWER:
                  // case 4.2.2
                  if (alt_to_X == 'USE_BLUE'
                      and alt_to_Y == 'USE_RED'):
                      return 'USE_RED_OR_BLUE'
                  if alt_to_X == 'USE_BLUE':
                      return 'USE_BLUE'
                  if alt_to_Y == 'USE_RED':
                      return 'USE_RED'
                  assert(False)
              else:
                  // case 4.2.3
                  if (F.HIGHER and not F.LOWER
                      and F.topo_order < A.topo_order):
                      return 'USE_RED'
                  else:
                      return 'USE_BLUE'
          else:
              // case 4.3
              if B.LOWER:
                  // case 4.3.1
                  if (F.LOWER and not F.HIGHER
                      and F.topo_order > B.topo_order):
                      return 'USE_BLUE'
                  else:
                      return 'USE_RED'
              if B.HIGHER:
                  // case 4.3.2
                  if (F.HIGHER and not F.LOWER
                      and F.topo_order < B.topo_order):
                      return 'USE_BLUE'
                  else:
                      return 'USE_RED'
              else:
                  // case 4.3.3
                  if A.topo_order < B.topo_order:
                      // case 4.3.3.1
                      if (alt_to_X == 'USE_BLUE'
                          and alt_to_Y == 'USE_RED'):
                          return 'USE_RED_OR_BLUE'
                      if alt_to_X == 'USE_BLUE':
                          return 'USE_BLUE'
                      if alt_to_Y == 'USE_RED':
                          return 'USE_RED'
                      assert(False)
```

```
                else:
                    // case 4.3.3.2
                    if (alt_to_X == 'USE_RED'
                        and alt_to_Y == 'USE_BLUE'):
                        return 'USE_RED_OR_BLUE'
                    if alt_to_X == 'USE_RED':
                        return 'USE_BLUE'
                    if alt_to_Y == 'USE_BLUE':
                        return 'USE_RED'
                    assert(False)
    assert(False)
```

Figure 29: Select_Alternates_Proxy_Node()

Select_Alternates_Proxy_Node(P,F,primary_intf) determines whether it is safe to use the blue path to P (which goes through X), the red path to P (which goes through Y), or either, when the primary_intf to node F (and possibly node F) fails. The basic approach is to run Select_Alternates(X,F,primary_intf) and Select_Alternates(Y,F,primary_intf) to determine which of the two MRT paths to X and which of the two MRT paths to Y is safe to use in the event of the failure of F. In general, we will find that if it is safe to use a particular path to X or Y when F fails, and Select_Proxy_Node_NHs() used that path when constructing the red or blue path to reach P, then it will also be safe to use that path to reach P when F fails. This rule has one exception which is covered below. First, we give a concrete example of how Select_Alternates_Proxy_Node() works in the common case.

The 21 ordering relationships used in Select_Proxy_Node_NHs() are repeated in Select_Alternates_Proxy_Node(). We focus on case 4.1.1 to give a detailed example of the reasoning used in Select_Alternates_Proxy_Node(). In Select_Proxy_Node_NHs(), we determined for case 4.1.1 that the red next hops to X and the blue next hops to Y allow us to reach X and Y by disjoint paths, and are thus the blue and red next hops to reach P. Therefore, if Select_Alternates(X, F, primary_intf) is run and we find that it is safe to USE_RED to reach X, then we also conclude that it is safe to use the MRT path through X to reach P (the blue path to P) when F fails. Similarly, if we run Select_Alternates(Y, F, primary_intf) and we find that it is safe to USE_BLUE to reach Y, then we also conclude that it is safe to use the MRT path through Y to reach P (the red path to P) when F fails. If both of the paths that were used in Select_Proxy_Node_NHs() to construct the blue and red paths to P are found to be safe to use to use to reach X and Y, t then we conclude that we can use either the red or the blue path to P.

This simple reasoning gives the correct answer in most of the cases. However, additional logic is needed when either A or B (but not both A and B) is unordered with respect to S. This applies to cases 4.1.3, 4.2.3, 4.3.1, and 4.3.2. Looking at case 4.1.3 in more detail, A is ordered less than S, but B is unordered with respect to S. In the discussion of case 4.1.3 above, we saw that Select_Proxy_Node_NHs() chose the red path to reach Y and the red path to reach X. We also saw that the red path to reach Y will follow the INCREASING next hops towards the localroot until it reaches some node (L) which is ordered greater than B, after which it will take DECREASING next hops to B. The problem is that the red path to reach P (the one that goes through Y) won't necessarily be the same as the red path to reach Y. This is because the next hop that node L computes for its red next hop to reach P may be different from the next hop it computes for its red next hop to reach Y. This is because B is ordered lower than L, so L applies case 4.1.2 of Select_Proxy_Node_NHs() in order to determine its next hops to reach P. If A.topo_order<B.topo_order (case 4.1.2.1), then L will choose DECREASING next hops directly to B, which is the same result that L computes in Compute_MRT_NextHops() to reach Y. However, if A.topo_order>B.topo_order (case 4.1.2.2), then L will choose INCREASING next hops to reach B, which is different from what L computes in Compute_MRT_NextHops() to reach Y. So, testing the safety of the path for S to reach Y on failure of F as a surrogate for the safety of using the red path to reach P is not reliable in this case. It is possible construct topologies where the red path to P hits F even though the red path to Y does not hit F.

Fortunately, there is enough information in the order relationships that we have already computed to still figure out which alternate to choose in these four cases. The basic idea is to always choose the path involving the ordered node, unless that path would hit F. Returning to case 4.1.3, we see that since A is ordered lower than S, the only way for S to hit F using a simple DECREASING path to A is for F to lie between A and S on the GADAG. This scenario is covered by requiring that F be lower than S (but not also higher than S) and that F.topo_order>A.topo_order in case 4.1.3.1.

We just need to confirm that it is safe to use the path involving B in this scenario. In case 4.1.3.1, either F is between A and S on the GADAG, or F is unordered with respect to A and lies on the DECREASING path from S to the localroot. When F is between A and S on the GADAG, then the path through B chosen to avoid A in Select_Proxy_Node_NHs() will also avoid F. When F is unordered with respect to A and lies on the DECREASING path from S to the localroot, then we consider two cases. Either F.topo_order<B.topo_order or F.topo_order>B.topo_order. In the first case, since F.topo_order<B.topo_order and F.topo_order>A.topo_order, it must be the case that A.topo_order<B.topo_order. Therefore, L will choose DECREASING next hops directly to B (case 4.1.2.1), which cannot hit F since F.topo_order<B.topo_order. In the second case, where F.topo_order>B.topo_order, the only way for the path involving B to hit F is if it DECREASES from L to B through F, i.e., it must be that L>>F>>B. However, since S>>F, this would imply that S>>B. However, we know that S is unordered with respect to B, so the second case cannot occur. So we have demonstrated that the red path to P (which goes via B and Y) is safe to use under the conditions of 4.1.3.1. Similar reasoning can be applied to the other three special cases where either A or B is unordered with respect to S.

6. MRT Lowpoint Algorithm: Next-Hop Conformance

This specification defines the MRT Lowpoint algorithm, which includes the construction of a common GADAG and the computation of MRT-Red and MRT-Blue next hops to each node in the graph. An implementation MAY select any subset of next hops for MRT-Red and MRT-Blue that respect the available nodes that are described in Section 5.7 for each of the MRT-Red and MRT-Blue and the selected next hops are further along in the interval of allowed nodes towards the destination.

For example, the MRT-Blue next hops used when the destination Y >> X, the computing router, MUST be one or more nodes, T, whose topo_order is in the interval [X.topo_order, Y.topo_order] and where Y >> T or Y is T. Similarly, the MRT-Red next hops MUST be have a topo_order in the interval [R-small.topo_order, X.topo_order] or [Y.topo_order, R-big.topo_order].

Implementations SHOULD implement the Select_Alternates() function to pick an MRT-FRR alternate.

7. Broadcast Interfaces

When broadcast interfaces are used to connect nodes, the broadcast network MUST be represented as a pseudonode, where each real node connects to the pseudonode. The interface metric in the direction from real node to pseudonode is the non-zero interface metric, while the interface metric in the direction from the pseudonode to the real node is set to zero. This is consistent with the way that broadcast interfaces are represented as pseudonodes in IS-IS and OSPF.

Pseudonodes MUST be treated as equivalent to real nodes in the network graph used in the MRT Lowpoint algorithm with a few exceptions detailed below.

The pseudonodes MUST be included in the computation of the GADAG. The neighbors of the pseudonode need to know the mrt_node_id of the pseudonode in order to consistently order interfaces, which is needed to compute the GADAG. The mrt_node_id for IS-IS is the 7-octet neighbor system ID and pseudonode number in TLV 22 or TLV 222. The mrt_node_id for OSPFv2 is the 4-octet interface address of the Designated Router found in the Link ID field for the link type 2 (transit network) in the Router-LSA. The mrt_node_id for OSPFv3 is the 4 octet interface address of the Designated Router found in the Neighbor Interface ID field for the link type 2 (transit network) in the Router-LSA. Note that this is different from the Neighbor Router ID field used for the mrt_node_id for point-to-point links in OSPFv3 Router-LSAs given in Figure 15.

Pseudonodes MUST NOT be considered candidates for selection as GADAG root. This rule is intended to result in a more stable network-wide selection of GADAG root by removing the possibility that the change of Designated Router or Designated Intermediate System on a broadcast network can result in a change of GADAG root.

7.1. Computing MRT Next Hops on Broadcast Networks

The pseudonode does not correspond to a real node, so it is not actually involved in forwarding. A real node on a broadcast network cannot simply forward traffic to the broadcast network. It must specify another real node on the broadcast network as the next hop. On a network graph where a broadcast network is represented by a pseudonode, this means that if a real node determines that the next hop to reach a given destination is a pseudonode, it must also determine the next-next-hop for that destination in the network graph, which corresponds to a real node attached to the broadcast network.

It is interesting to note that this issue is not unique to the MRT algorithm, but is also encountered in normal SPF computations for IGPs. Section 16.1.1 of [RFC2328] describes how this is done for OSPF. When OSPF runs its shortest-path tree calculation, it deals with pseudonodes in the following manner. Whenever the calculating router finds a shorter path to reach a real destination node and the shorter path to the destination is a single pseudonode hop, then the next hop for that destination is taken from the interface IP address in the Router-LSA correspond to the link to the real destination node.

For IS-IS, in the example pseudocode implementation of Dijkstra's algorithm in Annex C of [ISO10589-Second-Edition], whenever the algorithm encounters an adjacency from a real node to a pseudonode, it gets converted to a set of adjacencies from the real node to the neighbors of the pseudonode. In this way, the computed next hops point all the way to the real node, and not the pseudonode.

We could avoid the problem of determining next hops across pseudonodes in MRT by converting the pseudonode representation of broadcast networks to a full mesh of links between real nodes on the same network. However, if we make that conversion before computing the GADAG, we lose information about which links actually correspond to a single physical interface into the broadcast network. This could result computing red and blue next hops that use the same broadcast interface, in which case neither the red nor the blue next hop would be usable as an alternate on failure of the broadcast interface.

Instead, we take the following approach, which maintains the property that either the red and blue next hop will avoid the broadcast network, if topologically allowed. We run the MRT Lowpoint algorithm treating the pseudonodes as equivalent to real nodes in the network graph, with the exceptions noted above. In addition to running the MRT Lowpoint algorithm from the point of view of itself, a computing router connected to a pseudonode MUST also run the MRT Lowpoint algorithm from the point of view of each of its pseudonode neighbors. For example, if a computing router S determines that its MRT red next hop to reach a destination D is a pseudonode P, S looks at its MRT Lowpoint algorithm computation from P's point of view to determine P's red next hop to reach D, say interface 1 on node X. S now knows that its real red next hop to reach D is interface 1 on node X on the broadcast network represented by P, and it can install the corresponding entry in its FIB.

7.2. Using MRT Next Hops as Alternates in the Event of Failures on Broadcast Networks

In the previous section, we specified how to compute MRT next hops when broadcast networks are involved. In this section, we discuss how a PLR can use those MRT next hops in the event of failures involving broadcast networks.

A PLR attached to a broadcast network running only OSPF or IS-IS with large Hello intervals has limited ability to quickly detect failures on a broadcast network. The only failure mode that can be quickly detected is the failure of the physical interface connecting the PLR to the broadcast network. For the failure of the interface connecting the PLR to the broadcast network, the alternate that avoids the broadcast network can be computed by using the broadcast network pseudonode as F, the primary next-hop node, in Select_Alternates(). This will choose an alternate path that avoids the broadcast network. However, the alternate path will not necessarily avoid all of the real nodes connected to the broadcast network. This is because we have used the pseudonode to represent the broadcast network. And we have enforced the node-protecting property of MRT on the pseudonode to provide protection against failure of the broadcast network, not the real next-hop nodes on the broadcast network. This is the best that we can hope to do if failure of the broadcast interface is the only failure mode that the PLR can respond to.

We can improve on this if the PLR also has the ability to quickly detect a lack of connectivity across the broadcast network to a given IP-layer node. This can be accomplished by running BFD between all pairs of IGP neighbors on the broadcast network. Note that in the case of OSPF, this would require establishing BFD sessions between all pairs of neighbors in the 2-WAY state. When the PLR can quickly detect the failure of a particular next hop across a broadcast network, the PLR can be more selective in its choice of alternates. For example, when the PLR observes that connectivity to an IP-layer node on a broadcast network has failed, the PLR may choose to still use the broadcast network to reach other IP-layer nodes that are still reachable. Or, if the PLR observes that connectivity has failed to several IP-layer nodes on the same broadcast network, it may choose to treat the entire broadcast network as failed. The choice of MRT alternates by a PLR for a particular set of failure conditions is a local decision, since it does not require coordination with other nodes.

8. Evaluation of Alternative Methods for Constructing GADAGs

This document specifies the MRT Lowpoint algorithm. One component of the algorithm involves constructing a common GADAG based on the network topology. The MRT Lowpoint algorithm computes the GADAG using the method described in Section 5.5. This method aims to minimize the amount of computation required to compute the GADAG. In the process of developing the MRT Lowpoint algorithm, two alternative methods for constructing GADAGs were also considered. These alternative methods are described in Appendices B and C. In general, these other two methods require more computation to compute the GADAG. The analysis below was performed to determine if the alternative GADAG construction methods produce shorter MRT alternate paths in real network topologies, and if so, to what extent.

Figure 30 compares results obtained using the three different methods for constructing GADAGs on five different service provider network topologies. MRT_LOWPOINT indicates the method specified in Section 5.5, while MRT_SPF and MRT_HYBRID indicate the methods specified in Appendices B and C, respectively. The columns on the right present the distribution of alternate path lengths for each GADAG construction method. Each MRT computation was performed using a same GADAG root chosen based on centrality.

For three of the topologies analyzed (T201, T206, and T211), the use of MRT_SPF or MRT_HYBRID methods does not appear to provide a significantly shorter alternate path lengths compared to the MRT_LOWPOINT method. However, for two of the topologies (T216 and T219), the use of the MRT_SPF method resulted in noticeably shorter alternate path lengths than the use of the MRT_LOWPOINT or MRT_HYBRID methods.

It was decided to use the MRT_LOWPOINT method to construct the GADAG in the algorithm specified in this document, in order to initially offer an algorithm with lower computational requirements. These results indicate that in the future it may be useful to evaluate and potentially specify other MRT Lowpoint algorithm variants that use different GADAG construction methods.

Topology name GADAG construction method	percentage of failure scenarios protected by an alternate N hops longer than the primary path								
	0-1	2-3	4-5	6-7	8-9	10 -11	12 -13	14 -15	no alt <16
T201(avg primary hops=3.5)									
MRT_HYBRID	33	26	23	6	3				
MRT_SPF	33	36	23	6	3				
MRT_LOWPOINT	33	36	23	6	3				
T206(avg primary hops=3.7)									
MRT_HYBRID	50	35	13	2					
MRT_SPF	50	35	13	2					
MRT_LOWPOINT	55	32	13						
T211(avg primary hops=3.3)									
MRT_HYBRID	86	14							
MRT_SPF	86	14							
MRT_LOWPOINT	85	15	1						
T216(avg primary hops=5.2)									
MRT_HYBRID	23	22	18	13	10	7	4	2	2
MRT_SPF	35	32	19	9	3	1			
MRT_LOWPOINT	28	25	18	11	7	6	3	2	1
T219(avg primary hops=7.7)									
MRT_HYBRID	20	16	13	10	7	5	5	5	3
MRT_SPF	31	23	19	12	7	4	2	1	
MRT_LOWPOINT	19	14	15	12	10	8	7	6	10

Figure 30: The Length of Alternate Paths for Various GADAG Construction Methods

9. Operational Considerations

This section discusses operational considerations related to the MRT Lowpoint algorithm and other potential MRT algorithm variants. For a discussion of operational considerations related to MRT-FRR in general, see the "Operational Considerations" section of [RFC7812].

9.1. GADAG Root Selection

The Default MRT Profile uses the GADAG Root Selection Priority advertised by routers as the primary criterion for selecting the GADAG root. It is RECOMMENDED that an operator designate a set of routers as good choices for selection as GADAG root by setting the GADAG Root Selection Priority for that set of routers to lower (more preferred) numerical values. Criteria for making this designation are discussed below.

Analysis has shown that the centrality of a router can have a significant impact on the lengths of the alternate paths computed. Therefore, it is RECOMMENDED that off-line analysis that considers the centrality of a router be used to help determine how good a choice a particular router is for the role of GADAG root.

If the router currently selected as GADAG root becomes unreachable in the IGP topology, then a new GADAG root will be selected. Changing the GADAG root can change the overall structure of the GADAG as well the paths of the red and MRT-Blue trees built using that GADAG. In order to minimize change in the associated red and MRT-Blue forwarding entries that can result from changing the GADAG root, it is RECOMMENDED that operators prioritize for selection as GADAG root those routers that are expected to consistently remain part of the IGP topology.

9.2. Destination-Rooted GADAGs

The MRT Lowpoint algorithm constructs a single GADAG rooted at a single node selected as the GADAG root. It is also possible to construct a different GADAG for each destination, with the GADAG rooted at the destination. A router can compute the MRT-Red and MRT-Blue next hops for that destination based on the GADAG rooted at that destination. Building a different GADAG for each destination is computationally more expensive, but it may give somewhat shorter alternate paths. Using destination-rooted GADAGs would require a new MRT profile to be created with a new MRT algorithm specification, since all routers in the MRT Island would need to use destination-rooted GADAGs.

10. Security Considerations

The algorithm described in this document does not introduce new security concerns beyond those already discussed in the document describing the MRT FRR architecture [RFC7812].

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC7812] Atlas, A., Bowers, C., and G. Enyedi, "An Architecture for IP/LDP Fast Reroute Using Maximally Redundant Trees (MRT-FRR)", RFC 7812, DOI 10.17487/RFC7812, June 2016, <http://www.rfc-editor.org/info/rfc7812>.

11.2. Informative References

[EnyediThesis] Enyedi, G., "Novel Algorithms for IP Fast Reroute", Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics Ph.D. Thesis, February 2011, <https://repozitorium.omikk.bme.hu/bitstream/handle/10890/1040/ertekezes.pdf>.

[IEEE8021Qca] IEEE, "IEEE Standard for Local and metropolitan area networks - Bridges and Bridged Networks - Amendment 24: Path Control and Reservation", IEEE 802.1Qca-2015, DOI 10.1109/IEEESTD.2016.7434544, 2016, <https://standards.ieee.org/findstds/standard/802.1Qca-2015.html>.

[ISO10589-Second-Edition] International Organization for Standardization, "Intermediate system to Intermediate system intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode Network Service (ISO 8473)", ISO/IEC 10589:2002, Second Edition, November 2002.

[Kahn_1962_topo_sort] Kahn, A., "Topological sorting of large networks", Communications of the ACM, Volume 5, Issue 11 DOI 10.1145/368996.369025, November 1962, <http://dl.acm.org/citation.cfm?doid=368996.369025>.

[MRTLinear]
Enyedi, G., Retvari, G., and A. Csaszar, "On Finding Maximally Redundant Trees in Strictly Linear Time", IEEE Symposium on Computers and Communications (ISCC), 2009, <http://opti.tmit.bme.hu/~enyedi/ipfrr/distMaxRedTree.pdf>.

[RFC2328] Moy, J., "OSPF Version 2", STD 54, RFC 2328, DOI 10.17487/RFC2328, April 1998, <http://www.rfc-editor.org/info/rfc2328>.

Appendix A. Python Implementation of MRT Lowpoint Algorithm

Below is Python code implementing the MRT Lowpoint algorithm specified in this document. The code is also posted on GitHub <https://github.com/cbowers/draft-ietf-rtgwg-mrt-frr-algorithm/blob/python_code_RFC7811/src/mrt_lowpoint_draft_text.py>.

While this Python code is believed to correctly implement the pseudocode description of the algorithm, in the event of a difference, the pseudocode description should be considered normative.

```
<CODE BEGINS>
# This program has been tested to run on Python 2.6 and 2.7
# (specifically Python 2.6.6 and 2.7.8 were tested).
# The program has known incompatibilities with Python 3.X.

# When executed, this program will generate a text file describing
# an example topology.  It then reads that text file back in as input
# to create the example topology, and runs the MRT Lowpoint algorithm.
# This was done to simplify the inclusion of the program as a single
# text file that can be extracted from the RFC.

# The output of the program is four text files containing a description
# of the GADAG, the blue and MRT-Reds for all destinations, and the
# MRT alternates for all failures.

import random
import os.path
import heapq

# simple Class definitions allow structure-like dot notation for
# variables and a convenient place to initialize those variables.
class Topology:
    def __init__(self):
        self.gadag_root = None
        self.node_list = []
        self.node_dict = {}
        self.test_gr = None
        self.island_node_list_for_test_gr = []
        self.stored_named_proxy_dict = {}
        self.init_new_computing_router()
    def init_new_computing_router(self):
        self.island_node_list = []
        self.named_proxy_dict = {}
```

```
class Node:
    def __init__(self):
        self.node_id = None
        self.intf_list = []
        self.profile_id_list = [0]
        self.GR_sel_priority = 128
        self.blue_next_hops_dict = {}
        self.red_next_hops_dict = {}
        self.blue_to_green_nh_dict = {}
        self.red_to_green_nh_dict = {}
        self.prefix_cost_dict = {}
        self.pnh_dict = {}
        self.alt_dict = {}
        self.init_new_computing_router()
    def init_new_computing_router(self):
        self.island_intf_list = []
        self.IN_MRT_ISLAND = False
        self.IN_GADAG = False
        self.dfs_number = None
        self.dfs_parent = None
        self.dfs_parent_intf = None
        self.dfs_child_list = []
        self.lowpoint_number = None
        self.lowpoint_parent = None
        self.lowpoint_parent_intf = None
        self.localroot = None
        self.block_id = None
        self.IS_CUT_VERTEX = False
        self.blue_next_hops = []
        self.red_next_hops = []
        self.primary_next_hops = []
        self.alt_list = []

class Interface:
    def __init__(self):
        self.metric = None
        self.area = None
        self.MRT_INELIGIBLE = False
        self.IGP_EXCLUDED = False
        self.SIMULATION_OUTGOING = False
        self.init_new_computing_router()
    def init_new_computing_router(self):
        self.UNDIRECTED = True
        self.INCOMING = False
        self.OUTGOING = False
        self.INCOMING_STORED = False
        self.OUTGOING_STORED = False
        self.IN_MRT_ISLAND = False
        self.PROCESSED = False

class Bundle:
    def __init__(self):
        self.UNDIRECTED = True
        self.OUTGOING = False
        self.INCOMING = False

class Alternate:
    def __init__(self):
        self.failed_intf = None
        self.red_or_blue = None
        self.nh_list = []
        self.fec = 'NO_ALTERNATE'
        self.prot = 'NO_PROTECTION'
        self.info = 'NONE'

class Proxy_Node_Attachment_Router:
    def __init__(self):
        self.prefix = None
        self.node = None
        self.named_proxy_cost = None
        self.min_lfin = None
        self.nh_intf_list = []

class Named_Proxy_Node:
    def __init__(self):
        self.node_id = None  #this is the prefix_id
        self.node_prefix_cost_list = []
        self.lfin_list = []
        self.pnar1 = None
        self.pnar2 = None
        self.pnar_X = None
        self.pnar_Y = None
        self.blue_next_hops = []
        self.red_next_hops = []
        self.primary_next_hops = []
        self.blue_next_hops_dict = {}
        self.red_next_hops_dict = {}
        self.pnh_dict = {}
        self.alt_dict = {}

def Interface_Compare(intf_a, intf_b):
    if intf_a.metric < intf_b.metric:
        return -1
    if intf_b.metric < intf_a.metric:
        return 1
    if intf_a.remote_node.node_id < intf_b.remote_node.node_id:
        return -1
    if intf_b.remote_node.node_id < intf_a.remote_node.node_id:
        return 1
    return 0

def Sort_Interfaces(topo):
    for node in topo.island_node_list:
        node.island_intf_list.sort(Interface_Compare)

def Reset_Computed_Node_and_Intf_Values(topo):
    topo.init_new_computing_router()
    for node in topo.node_list:
        node.init_new_computing_router()
        for intf in node.intf_list:
            intf.init_new_computing_router()

# This function takes a file with links represented by 2-digit
# numbers in the format:
# 01,05,10
# 05,02,30
# 02,01,15
# which represents a triangle topology with nodes 01, 05, and 02
# and symmetric metrics of 10, 30, and 15.

# Inclusion of a fourth column makes the metrics for the link
# asymmetric.  An entry of:
# 02,07,10,15
# creates a link from node 02 to 07 with metrics 10 and 15.
def Create_Topology_From_File(filename):
    topo = Topology()
    node_id_set= set()
    cols_list = []
    # on first pass just create nodes
    with open(filename + '.csv') as topo_file:
        for line in topo_file:
            line = line.rstrip('\r\n')
            cols=line.split(',')
            cols_list.append(cols)
            nodea_node_id = int(cols[0])
            nodeb_node_id = int(cols[1])
            if (nodea_node_id > 999 or nodeb_node_id > 999):
                print("node_id must be between 0 and 999.")
                print("exiting.")
                exit()
            node_id_set.add(nodea_node_id)
            node_id_set.add(nodeb_node_id)
    for node_id in node_id_set:
        node = Node()
```

```
            node.node_id = node_id
            topo.node_list.append(node)
            topo.node_dict[node_id] = node
        # on second pass create interfaces
        for cols in cols_list:
            nodea_node_id = int(cols[0])
            nodeb_node_id = int(cols[1])
            metric = int(cols[2])
            reverse_metric = int(cols[2])
            if len(cols) > 3:
                reverse_metric=int(cols[3])
            nodea = topo.node_dict[nodea_node_id]
            nodeb = topo.node_dict[nodeb_node_id]
            nodea_intf = Interface()
            nodea_intf.metric = metric
            nodea_intf.area = 0
            nodeb_intf = Interface()
            nodeb_intf.metric = reverse_metric
            nodeb_intf.area = 0
            nodea_intf.remote_intf = nodeb_intf
            nodeb_intf.remote_intf = nodea_intf
            nodea_intf.remote_node = nodeb
            nodeb_intf.remote_node = nodea
            nodea_intf.local_node = nodea
            nodeb_intf.local_node = nodeb
            nodea_intf.link_data = len(nodea.intf_list)
            nodeb_intf.link_data = len(nodeb.intf_list)
            nodea.intf_list.append(nodea_intf)
            nodeb.intf_list.append(nodeb_intf)
        return topo

def MRT_Island_Identification(topo, computing_rtr, profile_id, area):
       if profile_id in computing_rtr.profile_id_list:
           computing_rtr.IN_MRT_ISLAND = True
           explore_list = [computing_rtr]
       else:
           return
       while explore_list != []:
           next_rtr = explore_list.pop()
           for intf in next_rtr.intf_list:
               if ( (not intf.IN_MRT_ISLAND)
                    and (not intf.MRT_INELIGIBLE)
                    and (not intf.remote_intf.MRT_INELIGIBLE)
                    and (not intf.IGP_EXCLUDED) and intf.area == area
                    and (profile_id in intf.remote_node.profile_id_list)):
                    intf.IN_MRT_ISLAND = True
                    intf.remote_intf.IN_MRT_ISLAND = True
                    if (not intf.remote_node.IN_MRT_ISLAND):
```

```
                    intf.remote_INTF.IN_MRT_ISLAND = True
                    explore_list.append(intf.remote_node)

def Compute_Island_Node_List_For_Test_GR(topo, test_gr):
    Reset_Computed_Node_and_Intf_Values(topo)
    topo.test_gr = topo.node_dict[test_gr]
    MRT_Island_Identification(topo, topo.test_gr, 0, 0)
    for node in topo.node_list:
        if node.IN_MRT_ISLAND:
            topo.island_node_list_for_test_gr.append(node)

def Set_Island_Intf_and_Node_Lists(topo):
    for node in topo.node_list:
        if node.IN_MRT_ISLAND:
            topo.island_node_list.append(node)
            for intf in node.intf_list:
                if intf.IN_MRT_ISLAND:
                    node.island_intf_list.append(intf)

global_dfs_number = None

def Lowpoint_Visit(x, parent, intf_p_to_x):
    global global_dfs_number
    x.dfs_number = global_dfs_number
    x.lowpoint_number = x.dfs_number
    global_dfs_number += 1
    x.dfs_parent = parent
    if intf_p_to_x == None:
        x.dfs_parent_intf = None
    else:
        x.dfs_parent_intf = intf_p_to_x.remote_intf
    x.lowpoint_parent = None
    if parent != None:
        parent.dfs_child_list.append(x)
    for intf in x.island_intf_list:
        if intf.remote_node.dfs_number == None:
            Lowpoint_Visit(intf.remote_node, x, intf)
            if intf.remote_node.lowpoint_number < x.lowpoint_number:
                x.lowpoint_number = intf.remote_node.lowpoint_number
                x.lowpoint_parent = intf.remote_node
                x.lowpoint_parent_intf = intf
        else:
            if intf.remote_node is not parent:
                if intf.remote_node.dfs_number < x.lowpoint_number:
                    x.lowpoint_number = intf.remote_node.dfs_number
                    x.lowpoint_parent = intf.remote_node
                    x.lowpoint_parent_intf = intf
```

```
def Run_Lowpoint(topo):
    global global_dfs_number
    global_dfs_number = 0
    Lowpoint_Visit(topo.gadag_root, None, None)

max_block_id = None

def Assign_Block_ID(x, cur_block_id):
    global max_block_id
    x.block_id = cur_block_id
    for c in x.dfs_child_list:
        if (c.localroot is x):
            max_block_id += 1
            Assign_Block_ID(c, max_block_id)
        else:
            Assign_Block_ID(c, cur_block_id)

def Run_Assign_Block_ID(topo):
    global max_block_id
    max_block_id = 0
    Assign_Block_ID(topo.gadag_root, max_block_id)

def Construct_Ear(x, stack, intf, ear_type):
    ear_list = []
    cur_intf = intf
    not_done = True
    while not_done:
        cur_intf.UNDIRECTED = False
        cur_intf.OUTGOING = True
        cur_intf.remote_intf.UNDIRECTED = False
        cur_intf.remote_intf.INCOMING = True
        if cur_intf.remote_node.IN_GADAG == False:
            cur_intf.remote_node.IN_GADAG = True
            ear_list.append(cur_intf.remote_node)
            if ear_type == 'CHILD':
                cur_intf = cur_intf.remote_node.lowpoint_parent_intf
            else:
                assert ear_type == 'NEIGHBOR'
                cur_intf = cur_intf.remote_node.dfs_parent_intf
        else:
            not_done = False

    if ear_type == 'CHILD' and cur_intf.remote_node is x:
        # x is a cut-vertex and the local root for the block
        # in which the ear is computed
        x.IS_CUT_VERTEX = True
        localroot = x
    else:
        # inherit local root from the end of the ear
        localroot = cur_intf.remote_node.localroot

    while ear_list != []:
        y = ear_list.pop()
        y.localroot = localroot
        stack.append(y)

def Construct_GADAG_via_Lowpoint(topo):
    gadag_root = topo.gadag_root
    gadag_root.IN_GADAG = True
    gadag_root.localroot = None
    stack = []
    stack.append(gadag_root)
    while stack != []:
        x = stack.pop()
        for intf in x.island_intf_list:
            if ( intf.remote_node.IN_GADAG == False
                 and intf.remote_node.dfs_parent is x ):
                Construct_Ear(x, stack, intf, 'CHILD' )
        for intf in x.island_intf_list:
            if (intf.remote_node.IN_GADAG == False
                and intf.remote_node.dfs_parent is not x):
                Construct_Ear(x, stack, intf, 'NEIGHBOR')

def Assign_Remaining_Lowpoint_Parents(topo):
    for node in topo.island_node_list:
        if ( node is not topo.gadag_root
            and node.lowpoint_parent == None ):
            node.lowpoint_parent = node.dfs_parent
            node.lowpoint_parent_intf = node.dfs_parent_intf
            node.lowpoint_number = node.dfs_parent.dfs_number

def Add_Undirected_Block_Root_Links(topo):
    for node in topo.island_node_list:
        if node.IS_CUT_VERTEX or node is topo.gadag_root:
            for intf in node.island_intf_list:
                if ( intf.remote_node.localroot is not node
                     or intf.PROCESSED ):
                    continue
                bundle_list = []
                bundle = Bundle()
                for intf2 in node.island_intf_list:
                    if intf2.remote_node is intf.remote_node:
                        bundle_list.append(intf2)
                        if not intf2.UNDIRECTED:
                            bundle.UNDIRECTED = False
                            if intf2.INCOMING:
```

```
                                bundle.INCOMING = True
                            if intf2.OUTGOING:
                                bundle.OUTGOING = True
                if bundle.UNDIRECTED:
                    for intf3 in bundle_list:
                        intf3.UNDIRECTED = False
                        intf3.remote_intf.UNDIRECTED = False
                        intf3.PROCESSED = True
                        intf3.remote_intf.PROCESSED = True
                        intf3.OUTGOING = True
                        intf3.remote_intf.INCOMING = True
                else:
                    if (bundle.OUTGOING and bundle.INCOMING):
                        for intf3 in bundle_list:
                            intf3.UNDIRECTED = False
                            intf3.remote_intf.UNDIRECTED = False
                            intf3.PROCESSED = True
                            intf3.remote_intf.PROCESSED = True
                            intf3.OUTGOING = True
                            intf3.INCOMING = True
                            intf3.remote_intf.INCOMING = True
                            intf3.remote_intf.OUTGOING = True
                    elif bundle.OUTGOING:
                        for intf3 in bundle_list:
                            intf3.UNDIRECTED = False
                            intf3.remote_intf.UNDIRECTED = False
                            intf3.PROCESSED = True
                            intf3.remote_intf.PROCESSED = True
                            intf3.OUTGOING = True
                            intf3.remote_intf.INCOMING = True
                    elif bundle.INCOMING:
                        for intf3 in bundle_list:
                            intf3.UNDIRECTED = False
                            intf3.remote_intf.UNDIRECTED = False
                            intf3.PROCESSED = True
                            intf3.remote_intf.PROCESSED = True
                            intf3.INCOMING = True
                            intf3.remote_intf.OUTGOING = True

def Modify_Block_Root_Incoming_Links(topo):
    for node in topo.island_node_list:
        if ( node.IS_CUT_VERTEX == True or node is topo.gadag_root ):
            for intf in node.island_intf_list:
                if intf.remote_node.localroot is node:
                    if intf.INCOMING:
                        intf.INCOMING = False
                        intf.INCOMING_STORED = True
                        intf.remote_intf.OUTGOING = False
                        intf.remote_intf.OUTGOING_STORED = True

def Revert_Block_Root_Incoming_Links(topo):
    for node in topo.island_node_list:
        if ( node.IS_CUT_VERTEX == True or node is topo.gadag_root ):
            for intf in node.island_intf_list:
                if intf.remote_node.localroot is node:
                    if intf.INCOMING_STORED:
                        intf.INCOMING = True
                        intf.remote_intf.OUTGOING = True
                        intf.INCOMING_STORED = False
                        intf.remote_intf.OUTGOING_STORED = False

def Run_Topological_Sort_GADAG(topo):
    Modify_Block_Root_Incoming_Links(topo)
    for node in topo.island_node_list:
        node.unvisited = 0
        for intf in node.island_intf_list:
            if (intf.INCOMING == True):
                node.unvisited += 1
    working_list = []
    topo_order_list = []
    working_list.append(topo.gadag_root)
    while working_list != []:
        y = working_list.pop(0)
        topo_order_list.append(y)
        for intf in y.island_intf_list:
            if ( intf.OUTGOING == True):
                intf.remote_node.unvisited -= 1
                if intf.remote_node.unvisited == 0:
                    working_list.append(intf.remote_node)
    next_topo_order = 1
    while topo_order_list != []:
        y = topo_order_list.pop(0)
        y.topo_order = next_topo_order
        next_topo_order += 1
    Revert_Block_Root_Incoming_Links(topo)

def Set_Other_Undirected_Links_Based_On_Topo_Order(topo):
    for node in topo.island_node_list:
        for intf in node.island_intf_list:
            if intf.UNDIRECTED:
                if node.topo_order < intf.remote_node.topo_order:
                    intf.OUTGOING = True
                    intf.UNDIRECTED = False
                    intf.remote_intf.INCOMING = True
                    intf.remote_intf.UNDIRECTED = False
                else:
                    intf.INCOMING = True
                    intf.UNDIRECTED = False
                    intf.remote_intf.OUTGOING = True
                    intf.remote_intf.UNDIRECTED = False

def Initialize_Temporary_Interface_Flags(topo):
    for node in topo.island_node_list:
        for intf in node.island_intf_list:
            intf.PROCESSED = False
            intf.INCOMING_STORED = False
            intf.OUTGOING_STORED = False

def Add_Undirected_Links(topo):
    Initialize_Temporary_Interface_Flags(topo)
    Add_Undirected_Block_Root_Links(topo)
    Run_Topological_Sort_GADAG(topo)
    Set_Other_Undirected_Links_Based_On_Topo_Order(topo)

def In_Common_Block(x,y):
    if (  (x.block_id == y.block_id)
          or ( x is y.localroot) or (y is x.localroot) ):
        return True
    return False

def Copy_List_Items(target_list, source_list):
    del target_list[:] # Python idiom to remove all elements of a list
    for element in source_list:
        target_list.append(element)

def Add_Item_To_List_If_New(target_list, item):
    if item not in target_list:
        target_list.append(item)

def Store_Results(y, direction):
    if direction == 'INCREASING':
        y.HIGHER = True
        Copy_List_Items(y.blue_next_hops, y.next_hops)
    if direction == 'DECREASING':
        y.LOWER = True
        Copy_List_Items(y.red_next_hops, y.next_hops)
    if direction == 'NORMAL_SPF':
        y.primary_spf_metric = y.spf_metric
        Copy_List_Items(y.primary_next_hops, y.next_hops)
    if direction == 'MRT_ISLAND_SPF':
        Copy_List_Items(y.mrt_island_next_hops, y.next_hops)
    if direction == 'COLLAPSED_SPF':
        y.collapsed_metric = y.spf_metric
        Copy_List_Items(y.collapsed_next_hops, y.next_hops)
```

```
# Note that the Python heapq function allows for duplicate items,
# so we use the 'spf_visited' property to only consider a node
# as min_node the first time it gets removed from the heap.
def SPF_No_Traverse_Block_Root(topo, spf_root, block_root, direction):
    spf_heap = []
    for y in topo.island_node_list:
        y.spf_metric = 2147483647 # 2^31-1
        y.next_hops = []
        y.spf_visited = False
    spf_root.spf_metric = 0
    heapq.heappush(spf_heap,
                   (spf_root.spf_metric, spf_root.node_id,  spf_root) )
    while spf_heap != []:
        #extract third element of tuple popped from heap
        min_node = heapq.heappop(spf_heap)[2]
        if min_node.spf_visited:
            continue
        min_node.spf_visited = True
        Store_Results(min_node, direction)
        if ( (min_node is spf_root) or (min_node is not block_root) ):
            for intf in min_node.island_intf_list:
                if ( ( (direction == 'INCREASING' and intf.OUTGOING )
                    or (direction == 'DECREASING' and intf.INCOMING ) )
                    and In_Common_Block(spf_root, intf.remote_node) ) :
                    path_metric = min_node.spf_metric + intf.metric
                    if path_metric < intf.remote_node.spf_metric:
                        intf.remote_node.spf_metric = path_metric
                        if min_node is spf_root:
                            intf.remote_node.next_hops = [intf]
                        else:
                            Copy_List_Items(intf.remote_node.next_hops,
                                            min_node.next_hops)
                        heapq.heappush(spf_heap,
                                       ( intf.remote_node.spf_metric,
                                         intf.remote_node.node_id,
                                         intf.remote_node ) )
                    elif path_metric == intf.remote_node.spf_metric:
                        if min_node is spf_root:
                            Add_Item_To_List_If_New(
                                intf.remote_node.next_hops,intf)
                        else:
                            for nh_intf in min_node.next_hops:
                                Add_Item_To_List_If_New(
                                    intf.remote_node.next_hops,nh_intf)

def Normal_SPF(topo, spf_root):
    spf_heap = []
    for y in topo.node_list:
        y.spf_metric = 2147483647 # 2^31-1 as max metric
        y.next_hops = []
        y.primary_spf_metric = 2147483647
        y.primary_next_hops = []
        y.spf_visited = False
    spf_root.spf_metric = 0
    heapq.heappush(spf_heap,
                   (spf_root.spf_metric,spf_root.node_id,spf_root) )
    while spf_heap != []:
        #extract third element of tuple popped from heap
        min_node = heapq.heappop(spf_heap)[2]
        if min_node.spf_visited:
            continue
        min_node.spf_visited = True
        Store_Results(min_node, 'NORMAL_SPF')
        for intf in min_node.intf_list:
            path_metric = min_node.spf_metric + intf.metric
            if path_metric < intf.remote_node.spf_metric:
                intf.remote_node.spf_metric = path_metric
                if min_node is spf_root:
                    intf.remote_node.next_hops = [intf]
                else:
                    Copy_List_Items(intf.remote_node.next_hops,
                                    min_node.next_hops)
                heapq.heappush(spf_heap,
                               ( intf.remote_node.spf_metric,
                                 intf.remote_node.node_id,
                                 intf.remote_node ) )
            elif path_metric == intf.remote_node.spf_metric:
                if min_node is spf_root:
                    Add_Item_To_List_If_New(
                        intf.remote_node.next_hops,intf)
                else:
                    for nh_intf in min_node.next_hops:
                        Add_Item_To_List_If_New(
                            intf.remote_node.next_hops,nh_intf)

def Set_Edge(y):
    if (y.blue_next_hops == [] and y.red_next_hops == []):
        Set_Edge(y.localroot)
        Copy_List_Items(y.blue_next_hops,y.localroot.blue_next_hops)
        Copy_List_Items(y.red_next_hops ,y.localroot.red_next_hops)
        y.order_proxy = y.localroot.order_proxy

def Compute_MRT_NH_For_One_Src_To_Island_Dests(topo,x):
    for y in topo.island_node_list:
        y.HIGHER = False
        y.LOWER = False
```

```
            y.red_next_hops = []
            y.blue_next_hops = []
            y.order_proxy = y
        SPF_No_Traverse_Block_Root(topo, x, x.localroot, 'INCREASING')
        SPF_No_Traverse_Block_Root(topo, x, x.localroot, 'DECREASING')
        for y in topo.island_node_list:
            if ( y is not x and (y.block_id == x.block_id) ):
                assert (not ( y is x.localroot or x is y.localroot) )
                assert(not (y.HIGHER and y.LOWER) )
                if y.HIGHER == True:
                    Copy_List_Items(y.red_next_hops,
                                    x.localroot.red_next_hops)
                elif y.LOWER == True:
                    Copy_List_Items(y.blue_next_hops,
                                    x.localroot.blue_next_hops)
                else:
                    Copy_List_Items(y.blue_next_hops,
                                    x.localroot.red_next_hops)
                    Copy_List_Items(y.red_next_hops,
                                    x.localroot.blue_next_hops)

        # Inherit x's MRT next hops to reach the GADAG root
        # from x's MRT next hops to reach its local root,
        # but first check if x is the gadag_root (in which case
        # x does not have a local root) or if x's local root
        # is the gadag root (in which case we already have the
        # x's MRT next hops to reach the gadag root)
        if x is not topo.gadag_root and x.localroot is not topo.gadag_root:
            Copy_List_Items(topo.gadag_root.blue_next_hops,
                            x.localroot.blue_next_hops)
            Copy_List_Items(topo.gadag_root.red_next_hops,
                            x.localroot.red_next_hops)
            topo.gadag_root.order_proxy = x.localroot

        # Inherit next hops and order_proxies to other blocks
        for y in topo.island_node_list:
            if (y is not topo.gadag_root and y is not x ):
                Set_Edge(y)

    def Store_MRT_Nexthops_For_One_Src_To_Island_Dests(topo,x):
        for y in topo.island_node_list:
            if y is x:
                continue
            x.blue_next_hops_dict[y.node_id] = []
            x.red_next_hops_dict[y.node_id] = []
            Copy_List_Items(x.blue_next_hops_dict[y.node_id],
                            y.blue_next_hops)
            Copy_List_Items(x.red_next_hops_dict[y.node_id],
                            y.red_next_hops)

    def Store_Primary_and_Alts_For_One_Src_To_Island_Dests(topo,x):
        for y in topo.island_node_list:
            x.pnh_dict[y.node_id] = []
            Copy_List_Items(x.pnh_dict[y.node_id], y.primary_next_hops)
            x.alt_dict[y.node_id] = []
            Copy_List_Items(x.alt_dict[y.node_id], y.alt_list)

    def Store_Primary_NHs_For_One_Source_To_Nodes(topo,x):
        for y in topo.node_list:
            x.pnh_dict[y.node_id] = []
            Copy_List_Items(x.pnh_dict[y.node_id], y.primary_next_hops)

    def Store_MRT_NHs_For_One_Src_To_Named_Proxy_Nodes(topo,x):
        for prefix in topo.named_proxy_dict:
            P = topo.named_proxy_dict[prefix]
            x.blue_next_hops_dict[P.node_id] = []
            x.red_next_hops_dict[P.node_id] = []
            Copy_List_Items(x.blue_next_hops_dict[P.node_id],
                            P.blue_next_hops)
            Copy_List_Items(x.red_next_hops_dict[P.node_id],
                            P.red_next_hops)

    def Store_Alts_For_One_Src_To_Named_Proxy_Nodes(topo,x):
        for prefix in topo.named_proxy_dict:
            P = topo.named_proxy_dict[prefix]
            x.alt_dict[P.node_id] = []
            Copy_List_Items(x.alt_dict[P.node_id],
                            P.alt_list)

    def Store_Primary_NHs_For_One_Src_To_Named_Proxy_Nodes(topo,x):
        for prefix in topo.named_proxy_dict:
            P = topo.named_proxy_dict[prefix]
            x.pnh_dict[P.node_id] = []
            Copy_List_Items(x.pnh_dict[P.node_id],
                            P.primary_next_hops)

    def Select_Alternates_Internal(D, F, primary_intf,
                                   D_lower, D_higher, D_topo_order):
        if D_higher and D_lower:
            if F.HIGHER and F.LOWER:
                if F.topo_order > D_topo_order:
                    return 'USE_BLUE'
                else:
                    return 'USE_RED'
            if F.HIGHER:
                return 'USE_RED'
            if F.LOWER:
                return 'USE_BLUE'
            assert(primary_intf.MRT_INELIGIBLE
                   or primary_intf.remote_intf.MRT_INELIGIBLE)
            return 'USE_RED_OR_BLUE'
        if D_higher:
            if F.HIGHER and F.LOWER:
                return 'USE_BLUE'
            if F.LOWER:
                return 'USE_BLUE'
            if F.HIGHER:
                if (F.topo_order > D_topo_order):
                    return 'USE_BLUE'
                if (F.topo_order < D_topo_order):
                    return 'USE_RED'
                assert(False)
            assert(primary_intf.MRT_INELIGIBLE
                   or primary_intf.remote_intf.MRT_INELIGIBLE)
            return 'USE_RED_OR_BLUE'
        if D_lower:
            if F.HIGHER and F.LOWER:
                return 'USE_RED'
            if F.HIGHER:
                return 'USE_RED'
            if F.LOWER:
                if F.topo_order > D_topo_order:
                    return 'USE_BLUE'
                if F.topo_order < D_topo_order:
                    return 'USE_RED'
                assert(False)
            assert(primary_intf.MRT_INELIGIBLE
                   or primary_intf.remote_intf.MRT_INELIGIBLE)
            return 'USE_RED_OR_BLUE'
        else: # D is unordered wrt S
            if F.HIGHER and F.LOWER:
                if primary_intf.OUTGOING and primary_intf.INCOMING:
                    # This can happen when F and D are in different blocks
                    return 'USE_RED_OR_BLUE'
                if primary_intf.OUTGOING:
                    return 'USE_BLUE'
                if primary_intf.INCOMING:
                    return 'USE_RED'
                #This can occur when primary_intf is MRT_INELIGIBLE.
                #This appears to be a case where the special
                #construction of the GADAG allows us to choose red,
                #whereas with an arbitrary GADAG, neither red nor blue
                #is guaranteed to work.
                assert(primary_intf.MRT_INELIGIBLE
                       or primary_intf.remote_intf.MRT_INELIGIBLE)
                return 'USE_RED'
            if F.LOWER:
                return 'USE_RED'
            if F.HIGHER:
                return 'USE_BLUE'
            assert(primary_intf.MRT_INELIGIBLE
                   or primary_intf.remote_intf.MRT_INELIGIBLE)
            if F.topo_order > D_topo_order:
                return 'USE_BLUE'
            else:
                return 'USE_RED'

    def Select_Alternates(D, F, primary_intf):
        S = primary_intf.local_node
        if not In_Common_Block(F, S):
            return 'PRIM_NH_IN_DIFFERENT_BLOCK'
        if (D is F) or (D.order_proxy is F):
            return 'PRIM_NH_IS_D_OR_OP_FOR_D'
        D_lower = D.order_proxy.LOWER
        D_higher = D.order_proxy.HIGHER
        D_topo_order = D.order_proxy.topo_order
        return Select_Alternates_Internal(D, F, primary_intf,
                                          D_lower, D_higher, D_topo_order)


    def Is_Remote_Node_In_NH_List(node, intf_list):
        for intf in intf_list:
            if node is intf.remote_node:
                return True
        return False

    def Select_Alts_For_One_Src_To_Island_Dests(topo,x):
        Normal_SPF(topo, x)
        for D in topo.island_node_list:
            D.alt_list = []
            if D is x:
                continue
            for failed_intf in D.primary_next_hops:
                alt = Alternate()
                alt.failed_intf = failed_intf
                cand_alt_list = []
                F = failed_intf.remote_node
                #We need to test if F is in the island, as opposed
                #to just testing if failed_intf is in island_intf_list,
                #because failed_intf could be marked as MRT_INELIGIBLE.
                if F in topo.island_node_list:
                    alt.info = Select_Alternates(D, F, failed_intf)
                else:
                    #The primary next hop is not in the MRT Island.
                    #Either red or blue will avoid the primary next hop,
                    #because the primary next hop is not even in the
                    #GADAG.
                    alt.info = 'USE_RED_OR_BLUE'

                if (alt.info == 'USE_RED_OR_BLUE'):
                    alt.red_or_blue = random.choice(['USE_RED','USE_BLUE'])
                if (alt.info == 'USE_BLUE'
                    or alt.red_or_blue == 'USE_BLUE'):
                    Copy_List_Items(alt.nh_list, D.blue_next_hops)
                    alt.fec = 'BLUE'
                    alt.prot = 'NODE_PROTECTION'
                if (alt.info == 'USE_RED' or alt.red_or_blue == 'USE_RED'):
                    Copy_List_Items(alt.nh_list, D.red_next_hops)
                    alt.fec = 'RED'
                    alt.prot = 'NODE_PROTECTION'
                if (alt.info == 'PRIM_NH_IN_DIFFERENT_BLOCK'):
                    alt.fec = 'NO_ALTERNATE'
                    alt.prot = 'NO_PROTECTION'
                if (alt.info == 'PRIM_NH_IS_D_OR_OP_FOR_D'):
                    if failed_intf.OUTGOING and failed_intf.INCOMING:
                        # cut-link: if there are parallel cut links, use
                        # the link(s) with lowest metric that are not
                        # primary intf or None
                        cand_alt_list = [None]
                        min_metric = 2147483647
                        for intf in x.island_intf_list:
                            if ( intf is not failed_intf and
                                 (intf.remote_node is
                                 failed_intf.remote_node)):
                                if intf.metric < min_metric:
                                    cand_alt_list = [intf]
                                    min_metric = intf.metric
                                elif intf.metric == min_metric:
                                    cand_alt_list.append(intf)
                        if cand_alt_list != [None]:
                            alt.fec = 'GREEN'
                            alt.prot = 'PARALLEL_CUTLINK'
                        else:
                            alt.fec = 'NO_ALTERNATE'
                            alt.prot = 'NO_PROTECTION'
                        Copy_List_Items(alt.nh_list, cand_alt_list)

                    # Is_Remote_Node_In_NH_List() is used, as opposed
                    # to just checking if failed_intf is in D.red_next_hops,
                    # because failed_intf could be marked as MRT_INELIGIBLE.
                    elif Is_Remote_Node_In_NH_List(F, D.red_next_hops):
                        Copy_List_Items(alt.nh_list, D.blue_next_hops)
                        alt.fec = 'BLUE'
                        alt.prot = 'LINK_PROTECTION'
                    elif Is_Remote_Node_In_NH_List(F, D.blue_next_hops):
                        Copy_List_Items(alt.nh_list, D.red_next_hops)
                        alt.fec = 'RED'
                        alt.prot = 'LINK_PROTECTION'
                    else:
                        alt.fec = random.choice(['RED','BLUE'])
                        alt.prot = 'LINK_PROTECTION'

                D.alt_list.append(alt)

    def Write_GADAG_To_File(topo, file_prefix):
        gadag_edge_list = []
        for node in topo.node_list:
            for intf in node.intf_list:
                if intf.SIMULATION_OUTGOING:
                    local_node =  "%04d" % (intf.local_node.node_id)
                    remote_node = "%04d" % (intf.remote_node.node_id)
                    intf_data = "%03d" % (intf.link_data)
                    edge_string=(local_node+','+remote_node+','+
                                 intf_data+'\n')
                    gadag_edge_list.append(edge_string)
        gadag_edge_list.sort();
        filename = file_prefix + '_gadag.csv'
        with open(filename, 'w') as gadag_file:
            gadag_file.write('local_node,'\
                             'remote_node,local_intf_link_data\n')
            for edge_string in gadag_edge_list:
                gadag_file.write(edge_string);

    def Write_MRTs_For_All_Dests_To_File(topo, color, file_prefix):
        edge_list = []
        for node in topo.island_node_list_for_test_gr:
            if color == 'blue':
                node_next_hops_dict = node.blue_next_hops_dict
            elif color == 'red':
                node_next_hops_dict = node.red_next_hops_dict
            for dest_node_id in node_next_hops_dict:
                for intf in node_next_hops_dict[dest_node_id]:
                    gadag_root =  "%04d" % (topo.gadag_root.node_id)
                    dest_node =  "%04d" % (dest_node_id)
                    local_node =  "%04d" % (intf.local_node.node_id)
                    remote_node = "%04d" % (intf.remote_node.node_id)
                    intf_data = "%03d" % (intf.link_data)
                    edge_string=(gadag_root+','+dest_node+','+local_node+
                                   ','+remote_node+','+intf_data+'\n')
                    edge_list.append(edge_string)
        edge_list.sort()
        filename = file_prefix + '_' + color + '_to_all.csv'
        with open(filename, 'w') as mrt_file:
            mrt_file.write('gadag_root,dest,'\
                'local_node,remote_node,link_data\n')
            for edge_string in edge_list:
                mrt_file.write(edge_string);

    def Write_Both_MRTs_For_All_Dests_To_File(topo, file_prefix):
        Write_MRTs_For_All_Dests_To_File(topo, 'blue', file_prefix)
        Write_MRTs_For_All_Dests_To_File(topo, 'red', file_prefix)

    def Write_Alternates_For_All_Dests_To_File(topo, file_prefix):
        edge_list = []
        for x in topo.island_node_list_for_test_gr:
            for dest_node_id in x.alt_dict:
                alt_list = x.alt_dict[dest_node_id]
                for alt in alt_list:
                    for alt_intf in alt.nh_list:
                        gadag_root =  "%04d" % (topo.gadag_root.node_id)
                        dest_node =  "%04d" % (dest_node_id)
                        prim_local_node =  \
                            "%04d" % (alt.failed_intf.local_node.node_id)
                        prim_remote_node = \
                            "%04d" % (alt.failed_intf.remote_node.node_id)
                        prim_intf_data = \
                            "%03d" % (alt.failed_intf.link_data)
                        if alt_intf == None:
                            alt_local_node = "None"
                            alt_remote_node = "None"
                            alt_intf_data = "None"
                        else:
                            alt_local_node = \
                                "%04d" % (alt_intf.local_node.node_id)
                            alt_remote_node = \
                                "%04d" % (alt_intf.remote_node.node_id)
                            alt_intf_data = \
                                "%03d" % (alt_intf.link_data)
                        edge_string = (gadag_root+','+dest_node+','+
                            prim_local_node+','+prim_remote_node+','+
                            prim_intf_data+','+alt_local_node+','+
                            alt_remote_node+','+alt_intf_data+','+
                            alt.fec +'\n')
                        edge_list.append(edge_string)
        edge_list.sort()
        filename = file_prefix + '_alts_to_all.csv'
        with open(filename, 'w') as alt_file:
            alt_file.write('gadag_root,dest,'\
                'prim_nh.local_node,prim_nh.remote_node,'\
                'prim_nh.link_data,alt_nh.local_node,'\
                'alt_nh.remote_node,alt_nh.link_data,'\
                'alt_nh.fec\n')
            for edge_string in edge_list:
                alt_file.write(edge_string);

    def Raise_GADAG_Root_Selection_Priority(topo,node_id):
        node = topo.node_dict[node_id]
        node.GR_sel_priority = 255

    def Lower_GADAG_Root_Selection_Priority(topo,node_id):
        node = topo.node_dict[node_id]
        node.GR_sel_priority = 128

    def GADAG_Root_Compare(node_a, node_b):
        if (node_a.GR_sel_priority > node_b.GR_sel_priority):
            return 1
        elif (node_a.GR_sel_priority < node_b.GR_sel_priority):
            return -1
        else:
            if node_a.node_id > node_b.node_id:
                return 1
            elif node_a.node_id < node_b.node_id:
                return -1

    def Set_GADAG_Root(topo,computing_router):
        gadag_root_list = []
        for node in topo.island_node_list:
            gadag_root_list.append(node)
        gadag_root_list.sort(GADAG_Root_Compare)
        topo.gadag_root = gadag_root_list.pop()

    def Add_Prefix_Advertisements_From_File(topo, filename):
        prefix_filename = filename + '.prefix'
        cols_list = []
        if not os.path.exists(prefix_filename):
            return
        with open(prefix_filename) as prefix_file:
            for line in prefix_file:
                line = line.rstrip('\r\n')
                cols=line.split(',')
                cols_list.append(cols)
                prefix_id = int(cols[0])
                if prefix_id < 2000 or prefix_id >2999:
                    print('skipping the following line of prefix file')
                    print('prefix id should be between 2000 and 2999')
                    print(line)
                    continue
                prefix_node_id = int(cols[1])
                prefix_cost = int(cols[2])
                advertising_node = topo.node_dict[prefix_node_id]
                advertising_node.prefix_cost_dict[prefix_id] = prefix_cost

    def Add_Prefixes_for_Non_Island_Nodes(topo):
        for node in topo.node_list:
            if node.IN_MRT_ISLAND:
                continue
            prefix_id = node.node_id + 1000
            node.prefix_cost_dict[prefix_id] = 0

    def Add_Profile_IDs_from_File(topo, filename):
        profile_filename = filename + '.profile'
        for node in topo.node_list:
            node.profile_id_list = []
        cols_list = []
        if os.path.exists(profile_filename):
            with open(profile_filename) as profile_file:
                for line in profile_file:
                    line = line.rstrip('\r\n')
                    cols=line.split(',')
                    cols_list.append(cols)
                    node_id = int(cols[0])
                    profile_id = int(cols[1])
                    this_node = topo.node_dict[node_id]
                    this_node.profile_id_list.append(profile_id)
        else:
            for node in topo.node_list:
                node.profile_id_list = [0]

    def Island_Marking_SPF(topo,spf_root):
        spf_root.isl_marking_spf_dict = {}
        for y in topo.node_list:
            y.spf_metric = 2147483647 # 2^31-1 as max metric
            y.PATH_HITS_ISLAND = False
            y.next_hops = []
            y.spf_visited = False
        spf_root.spf_metric = 0
        spf_heap = []
        heapq.heappush(spf_heap,
                       (spf_root.spf_metric,spf_root.node_id,spf_root) )
        while spf_heap != []:
            #extract third element of tuple popped from heap
            min_node = heapq.heappop(spf_heap)[2]
            if min_node.spf_visited:
                continue
            min_node.spf_visited = True
            spf_root.isl_marking_spf_dict[min_node.node_id] = \
                (min_node.spf_metric, min_node.PATH_HITS_ISLAND)
            for intf in min_node.intf_list:
                path_metric = min_node.spf_metric + intf.metric
                if path_metric < intf.remote_node.spf_metric:
                    intf.remote_node.spf_metric = path_metric
                    if min_node is spf_root:
                        intf.remote_node.next_hops = [intf]
                    else:
                        Copy_List_Items(intf.remote_node.next_hops,
                                        min_node.next_hops)
                    if (intf.remote_node.IN_MRT_ISLAND):
                        intf.remote_node.PATH_HITS_ISLAND = True
                    else:
                        intf.remote_node.PATH_HITS_ISLAND = \
                            min_node.PATH_HITS_ISLAND
                    heapq.heappush(spf_heap,
                                   ( intf.remote_node.spf_metric,
                                     intf.remote_node.node_id,
                                     intf.remote_node ) )
                elif path_metric == intf.remote_node.spf_metric:
                    if min_node is spf_root:
                        Add_Item_To_List_If_New(
                            intf.remote_node.next_hops,intf)
                    else:
                        for nh_intf in min_node.next_hops:
                            Add_Item_To_List_If_New(
                                intf.remote_node.next_hops,nh_intf)
                    if (intf.remote_node.IN_MRT_ISLAND):
                        intf.remote_node.PATH_HITS_ISLAND = True
                    else:
                        if (intf.remote_node.PATH_HITS_ISLAND
                            or min_node.PATH_HITS_ISLAND):
                            intf.remote_node.PATH_HITS_ISLAND = True

    def Create_Basic_Named_Proxy_Nodes(topo):
        for node in topo.node_list:
            for prefix in node.prefix_cost_dict:
                prefix_cost = node.prefix_cost_dict[prefix]
                if prefix in topo.named_proxy_dict:
                    P = topo.named_proxy_dict[prefix]
                    P.node_prefix_cost_list.append((node,prefix_cost))
                else:
                    P = Named_Proxy_Node()
                    topo.named_proxy_dict[prefix] = P
                    P.node_id = prefix
                    P.node_prefix_cost_list = [(node,prefix_cost)]

    def Compute_Loop_Free_Island_Neighbors_For_Each_Prefix(topo):
        topo.island_nbr_set = set()
        topo.island_border_set = set()
        for node in topo.node_list:
            if node.IN_MRT_ISLAND:
                continue
            for intf in node.intf_list:
                if intf.remote_node.IN_MRT_ISLAND:
                    topo.island_nbr_set.add(node)
                    topo.island_border_set.add(intf.remote_node)

        for island_nbr in topo.island_nbr_set:
            Island_Marking_SPF(topo,island_nbr)

        for prefix in topo.named_proxy_dict:
            P = topo.named_proxy_dict[prefix]
            P.lfin_list = []
            for island_nbr in topo.island_nbr_set:
                min_isl_nbr_to_pref_cost = 2147483647
                for (adv_node, prefix_cost) in P.node_prefix_cost_list:
                    (adv_node_cost, path_hits_island) = \
                        island_nbr.isl_marking_spf_dict[adv_node.node_id]
                    isl_nbr_to_pref_cost = adv_node_cost + prefix_cost
                    if isl_nbr_to_pref_cost < min_isl_nbr_to_pref_cost:
                        min_isl_nbr_to_pref_cost = isl_nbr_to_pref_cost
                        min_path_hits_island = path_hits_island
                    elif isl_nbr_to_pref_cost == min_isl_nbr_to_pref_cost:
                        if min_path_hits_island or path_hits_island:
                            min_path_hits_island = True
                if not min_path_hits_island:
                    P.lfin_list.append( (island_nbr,
                                         min_isl_nbr_to_pref_cost) )


    def Compute_Island_Border_Router_LFIN_Pairs_For_Each_Prefix(topo):
        for ibr in topo.island_border_set:
            ibr.prefix_lfin_dict = {}
            ibr.min_intf_metric_dict = {}
            ibr.min_intf_list_dict = {}
            ibr.min_intf_list_dict[None] = None
            for intf in ibr.intf_list:
                if not intf.remote_node in topo.island_nbr_set:
                    continue
                if not intf.remote_node in ibr.min_intf_metric_dict:
                    ibr.min_intf_metric_dict[intf.remote_node] = \
                        intf.metric
                    ibr.min_intf_list_dict[intf.remote_node] = [intf]
                else:
                    if (intf.metric
                        < ibr.min_intf_metric_dict[intf.remote_node]):
                        ibr.min_intf_metric_dict[intf.remote_node] = \
                             intf.metric
                        ibr.min_intf_list_dict[intf.remote_node] = [intf]
                    elif (intf.metric
                          < ibr.min_intf_metric_dict[intf.remote_node]):
                        ibr.min_intf_list_dict[intf.remote_node].\
                            append(intf)

        for prefix in topo.named_proxy_dict:
            P = topo.named_proxy_dict[prefix]
            for ibr in topo.island_border_set:
                min_ibr_lfin_pref_cost = 2147483647
                min_lfin = None
                for (lfin, lfin_to_pref_cost) in P.lfin_list:
                    if not lfin in ibr.min_intf_metric_dict:
                        continue
                    ibr_lfin_pref_cost = \
                        ibr.min_intf_metric_dict[lfin] + lfin_to_pref_cost
                    if ibr_lfin_pref_cost < min_ibr_lfin_pref_cost:
                        min_ibr_lfin_pref_cost = ibr_lfin_pref_cost
                        min_lfin = lfin
                ibr.prefix_lfin_dict[prefix] = (min_lfin,
                    min_ibr_lfin_pref_cost,
                    ibr.min_intf_list_dict[min_lfin])

    def Proxy_Node_Att_Router_Compare(pnar_a, pnar_b):
        if pnar_a.named_proxy_cost < pnar_b.named_proxy_cost:
            return -1
        if pnar_b.named_proxy_cost < pnar_a.named_proxy_cost:
            return 1
        if pnar_a.node.node_id < pnar_b.node.node_id:
            return -1
        if pnar_b.node.node_id < pnar_a.node.node_id:
            return 1
        if pnar_a.min_lfin == None:
            return -1
        if pnar_b.min_lfin == None:
            return 1

    def Choose_Proxy_Node_Attachment_Routers(topo):
        for prefix in topo.named_proxy_dict:
            P = topo.named_proxy_dict[prefix]
            pnar_candidate_list = []
            for (node, prefix_cost) in P.node_prefix_cost_list:
                if not node.IN_MRT_ISLAND:
                    continue
                pnar = Proxy_Node_Attachment_Router()
                pnar.prefix = prefix
                pnar.named_proxy_cost = prefix_cost
                pnar.node = node
                pnar_candidate_list.append(pnar)
            for ibr in topo.island_border_set:
                (min_lfin, prefix_cost, min_intf_list) = \
                    ibr.prefix_lfin_dict[prefix]
                if min_lfin == None:
                    continue
                pnar = Proxy_Node_Attachment_Router()
                pnar.named_proxy_cost = prefix_cost
                pnar.node = ibr
                pnar.min_lfin = min_lfin
                pnar.nh_intf_list = min_intf_list
                pnar_candidate_list.append(pnar)
            pnar_candidate_list.sort(cmp=Proxy_Node_Att_Router_Compare)
            #pop first element from list
            first_pnar = pnar_candidate_list.pop(0)
            second_pnar = None
            for next_pnar in pnar_candidate_list:
                if next_pnar.node is first_pnar.node:
                    continue
                second_pnar = next_pnar
                break

            P.pnar1 = first_pnar
            P.pnar2 = second_pnar

    def Attach_Named_Proxy_Nodes(topo):
        Compute_Loop_Free_Island_Neighbors_For_Each_Prefix(topo)
        Compute_Island_Border_Router_LFIN_Pairs_For_Each_Prefix(topo)
        Choose_Proxy_Node_Attachment_Routers(topo)

    def Select_Proxy_Node_NHs(P,S):
        if P.pnar1.node.node_id < P.pnar2.node.node_id:
            X = P.pnar1.node
            Y = P.pnar2.node
        else:
            X = P.pnar2.node
            Y = P.pnar1.node
        P.pnar_X = X
        P.pnar_Y = Y
        A = X.order_proxy
        B = Y.order_proxy
        if (A is S.localroot
            and B is S.localroot):
            #print("1.0")
            Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
            Copy_List_Items(P.red_next_hops, Y.red_next_hops)
            return
        if (A is S.localroot
            and B is not S.localroot):
            #print("2.0")
            if B.LOWER:
                #print("2.1")
                Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                return
            if B.HIGHER:
                #print("2.2")
                Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                return
            else:
                #print("2.3")
                Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                return
        if (A is not S.localroot
            and B is S.localroot):
            #print("3.0")
            if A.LOWER:
                #print("3.1")
                Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                return
            if A.HIGHER:
                #print("3.2")
                Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                return
            else:
                #print("3.3")
                Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                return
        if (A is not S.localroot
            and B is not S.localroot):
            #print("4.0")
            if (S is A.localroot or S is B.localroot):
                #print("4.05")
                if A.topo_order < B.topo_order:
                    #print("4.05.1")
                    Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                    Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                    return
                else:
                    #print("4.05.2")
                    Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                    Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                    return
            if A.LOWER:
                #print("4.1")
                if B.HIGHER:
                    #print("4.1.1")
                    Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                    Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                    return
                if B.LOWER:
                    #print("4.1.2")
                    if A.topo_order < B.topo_order:
                        #print("4.1.2.1")
                        Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                        Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                        return
                    else:
                        #print("4.1.2.2")
                        Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                        Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                        return
                else:
                    #print("4.1.3")
                    Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                    Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                    return
            if A.HIGHER:
                #print("4.2")
                if B.HIGHER:
                    #print("4.2.1")
                    if A.topo_order < B.topo_order:
                        #print("4.2.1.1")
                        Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                        Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                        return
                    else:
                        #print("4.2.1.2")
                        Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                        Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                        return
                if B.LOWER:
                    #print("4.2.2")
                    Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                    Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                    return
                else:
                    #print("4.2.3")
                    Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                    Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                    return
            else:
                #print("4.3")
                if B.LOWER:
                    #print("4.3.1")
                    Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                    Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                    return
                if B.HIGHER:
                    #print("4.3.2")
                    Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                    Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                    return
                else:
                    #print("4.3.3")
                    if A.topo_order < B.topo_order:
                        #print("4.3.3.1")
                        Copy_List_Items(P.blue_next_hops, X.blue_next_hops)
                        Copy_List_Items(P.red_next_hops, Y.red_next_hops)
                        return
                    else:
                        #print("4.3.3.2")
                        Copy_List_Items(P.blue_next_hops, X.red_next_hops)
                        Copy_List_Items(P.red_next_hops, Y.blue_next_hops)
                        return
        assert(False)

    def Compute_MRT_NHs_For_One_Src_To_Named_Proxy_Nodes(topo,S):
        for prefix in topo.named_proxy_dict:
            P = topo.named_proxy_dict[prefix]
            if P.pnar2 == None:
                if S is P.pnar1.node:
                    # set the MRT next hops for the PNAR to
                    # reach the LFIN and change FEC to green
                    Copy_List_Items(P.blue_next_hops,
                                    P.pnar1.nh_intf_list)
                    S.blue_to_green_nh_dict[P.node_id] = True
                    Copy_List_Items(P.red_next_hops,
                                    P.pnar1.nh_intf_list)
                    S.red_to_green_nh_dict[P.node_id] = True
                else:
                    # inherit MRT NHs for P from pnar1
                    Copy_List_Items(P.blue_next_hops,
                                    P.pnar1.node.blue_next_hops)
                    Copy_List_Items(P.red_next_hops,
                                    P.pnar1.node.red_next_hops)
            else:
                Select_Proxy_Node_NHs(P,S)
                # set the MRT next hops for the PNAR to reach the LFIN
                # and change FEC to green rely on the red or blue
                # next hops being empty to figure out which one needs
                # to point to the LFIN.
                if S is P.pnar1.node:
                    this_pnar = P.pnar1
                elif S is P.pnar2.node:
                    this_pnar = P.pnar2
                else:
                    continue
                if P.blue_next_hops == []:
                    Copy_List_Items(P.blue_next_hops,
                        this_pnar.nh_intf_list)
                    S.blue_to_green_nh_dict[P.node_id] = True
                if P.red_next_hops == []:
                    Copy_List_Items(P.red_next_hops,
                        this_pnar.nh_intf_list)
                    S.red_to_green_nh_dict[P.node_id] = True

    def Select_Alternates_Proxy_Node(P,F,primary_intf):
        S = primary_intf.local_node
        X = P.pnar_X
        Y = P.pnar_Y
        A = X.order_proxy
        B = Y.order_proxy
        if F is A and F is B:
            return 'PRIM_NH_IS_OP_FOR_BOTH_X_AND_Y'
        if F is A:
            return 'USE_RED'
        if F is B:
            return 'USE_BLUE'

        if not In_Common_Block(A, B):
            if In_Common_Block(F, A):
                return 'USE_RED'
            elif In_Common_Block(F, B):
                return 'USE_BLUE'
            else:
                return 'USE_RED_OR_BLUE'
        if (not In_Common_Block(F, A)
            and not In_Common_Block(F, A) ):
            return 'USE_RED_OR_BLUE'

        alt_to_X = Select_Alternates(X, F, primary_intf)
        alt_to_Y = Select_Alternates(Y, F, primary_intf)

        if (alt_to_X == 'USE_RED_OR_BLUE'
            and alt_to_Y == 'USE_RED_OR_BLUE'):
            return 'USE_RED_OR_BLUE'
        if alt_to_X == 'USE_RED_OR_BLUE':
            return 'USE_BLUE'
        if alt_to_Y == 'USE_RED_OR_BLUE':
            return 'USE_RED'

        if (A is S.localroot
            and B is S.localroot):
            #print("1.0")
            if (alt_to_X == 'USE_BLUE' and alt_to_Y == 'USE_RED'):
                return 'USE_RED_OR_BLUE'
            if alt_to_X == 'USE_BLUE':
                return 'USE_BLUE'
            if alt_to_Y == 'USE_RED':
                return 'USE_RED'
            assert(False)
        if (A is S.localroot
            and B is not S.localroot):
            #print("2.0")
            if B.LOWER:
                #print("2.1")
                if (alt_to_X == 'USE_BLUE' and alt_to_Y == 'USE_RED'):
                    return 'USE_RED_OR_BLUE'
                if alt_to_X == 'USE_BLUE':
                    return 'USE_BLUE'
                if alt_to_Y == 'USE_RED':
                    return 'USE_RED'
                assert(False)
            if B.HIGHER:
                #print("2.2")
                if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_BLUE'):
                    return 'USE_RED_OR_BLUE'
                if alt_to_X == 'USE_RED':
                    return 'USE_BLUE'
                if alt_to_Y == 'USE_BLUE':
                    return 'USE_RED'
                assert(False)
            else:
                #print("2.3")
                if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_RED'):
                    return 'USE_RED_OR_BLUE'
                if alt_to_X == 'USE_RED':
                    return 'USE_BLUE'
                if alt_to_Y == 'USE_RED':
                    return 'USE_RED'
                assert(False)
        if (A is not S.localroot
            and B is S.localroot):
            #print("3.0")
            if A.LOWER:
                #print("3.1")
                if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_BLUE'):
                    return 'USE_RED_OR_BLUE'
                if alt_to_X == 'USE_RED':
                    return 'USE_BLUE'
                if alt_to_Y == 'USE_BLUE':
                    return 'USE_RED'
                assert(False)
            if A.HIGHER:
                #print("3.2")
                if (alt_to_X == 'USE_BLUE' and alt_to_Y == 'USE_RED'):
                    return 'USE_RED_OR_BLUE'
                if alt_to_X == 'USE_BLUE':
                    return 'USE_BLUE'
                if alt_to_Y == 'USE_RED':
                    return 'USE_RED'
                assert(False)
            else:
                #print("3.3")
                if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_RED'):
                    return 'USE_RED_OR_BLUE'
                if alt_to_X == 'USE_RED':
                    return 'USE_BLUE'
                if alt_to_Y == 'USE_RED':
                    return 'USE_RED'
                assert(False)
        if (A is not S.localroot
            and B is not S.localroot):
            #print("4.0")
            if (S is A.localroot or S is B.localroot):
                #print("4.05")
                if A.topo_order < B.topo_order:
                    #print("4.05.1")
                    if (alt_to_X == 'USE_BLUE' and alt_to_Y == 'USE_RED'):
                        return 'USE_RED_OR_BLUE'
                    if alt_to_X == 'USE_BLUE':
                        return 'USE_BLUE'
                    if alt_to_Y == 'USE_RED':
                        return 'USE_RED'
                    assert(False)
                else:
                    #print("4.05.2")
                    if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_BLUE'):
                        return 'USE_RED_OR_BLUE'
                    if alt_to_X == 'USE_RED':
                        return 'USE_BLUE'
                    if alt_to_Y == 'USE_BLUE':
                        return 'USE_RED'
                    assert(False)
            if A.LOWER:
                #print("4.1")
                if B.HIGHER:
                    #print("4.1.1")
                    if (alt_to_X == 'USE_RED' and alt_to_Y == 'USE_BLUE'):
                        return 'USE_RED_OR_BLUE'
                    if alt_to_X == 'USE_RED':
                        return 'USE_BLUE'
                    if alt_to_Y == 'USE_BLUE':
                        return 'USE_RED'
                    assert(False)
                if B.LOWER:
                    #print("4.1.2")
                    if A.topo_order < B.topo_order:
                        #print("4.1.2.1")
                        if (alt_to_X == 'USE_BLUE'
                            and alt_to_Y == 'USE_RED'):
                            return 'USE_RED_OR_BLUE'
                        if alt_to_X == 'USE_BLUE':
                            return 'USE_BLUE'
                        if alt_to_Y == 'USE_RED':
                            return 'USE_RED'
                        assert(False)
                    else:
                        #print("4.1.2.2")
                        if (alt_to_X == 'USE_RED'
                            and alt_to_Y == 'USE_BLUE'):
                            return 'USE_RED_OR_BLUE'
                        if alt_to_X == 'USE_RED':
                            return 'USE_BLUE'
                        if alt_to_Y == 'USE_BLUE':
                            return 'USE_RED'
                        assert(False)
                else:
                    #print("4.1.3")
                    if (F.LOWER and not F.HIGHER
                        and F.topo_order > A.topo_order):
                        #print("4.1.3.1")
                        return 'USE_RED'
                    else:
                        #print("4.1.3.2")
                        return 'USE_BLUE'
            if A.HIGHER:
                #print("4.2")
                if B.HIGHER:
                    #print("4.2.1")
                    if A.topo_order < B.topo_order:
                        #print("4.2.1.1")
                        if (alt_to_X == 'USE_BLUE'
                            and alt_to_Y == 'USE_RED'):
                            return 'USE_RED_OR_BLUE'
                        if alt_to_X == 'USE_BLUE':
                            return 'USE_BLUE'
                        if alt_to_Y == 'USE_RED':
                            return 'USE_RED'
                        assert(False)
                    else:
                        #print("4.2.1.2")
                        if (alt_to_X == 'USE_RED'
                            and alt_to_Y == 'USE_BLUE'):
                            return 'USE_RED_OR_BLUE'
                        if alt_to_X == 'USE_RED':
                            return 'USE_BLUE'
                        if alt_to_Y == 'USE_BLUE':
                            return 'USE_RED'
                        assert(False)
                if B.LOWER:
                    #print("4.2.2")
                    if (alt_to_X == 'USE_BLUE'
                        and alt_to_Y == 'USE_RED'):
                        return 'USE_RED_OR_BLUE'
                    if alt_to_X == 'USE_BLUE':
                        return 'USE_BLUE'
                    if alt_to_Y == 'USE_RED':
                        return 'USE_RED'
                    assert(False)
                else:
                    #print("4.2.3")
                    if (F.HIGHER and not F.LOWER
                        and F.topo_order < A.topo_order):
                        return 'USE_RED'
                    else:
                        return 'USE_BLUE'
            else:
                #print("4.3")
                if B.LOWER:
                    #print("4.3.1")
                    if (F.LOWER and not F.HIGHER
                        and F.topo_order > B.topo_order):
                        return 'USE_BLUE'
                    else:
                        return 'USE_RED'
                if B.HIGHER:
                    #print("4.3.2")
                    if (F.HIGHER and not F.LOWER
                        and F.topo_order < B.topo_order):
                        return 'USE_BLUE'
                    else:
                        return 'USE_RED'
                else:
                    #print("4.3.3")
                    if A.topo_order < B.topo_order:
                        #print("4.3.3.1")
                        if (alt_to_X == 'USE_BLUE'
                            and alt_to_Y == 'USE_RED'):
                            return 'USE_RED_OR_BLUE'
                        if alt_to_X == 'USE_BLUE':
                            return 'USE_BLUE'
                        if alt_to_Y == 'USE_RED':
                            return 'USE_RED'
                        assert(False)
                    else:
                        #print("4.3.3.2")
                        if (alt_to_X == 'USE_RED'
                            and alt_to_Y == 'USE_BLUE'):
                            return 'USE_RED_OR_BLUE'
                        if alt_to_X == 'USE_RED':
                            return 'USE_BLUE'
                        if alt_to_Y == 'USE_BLUE':
                            return 'USE_RED'
                        assert(False)
        assert(False)

    def Compute_Primary_NHs_For_One_Src_To_Named_Proxy_Nodes(topo,src):
        for prefix in topo.named_proxy_dict:
            P = topo.named_proxy_dict[prefix]
            min_total_pref_cost = 2147483647
            for (adv_node, prefix_cost) in P.node_prefix_cost_list:
                total_pref_cost = (adv_node.primary_spf_metric
                                   + prefix_cost)
                if total_pref_cost < min_total_pref_cost:
                    min_total_pref_cost = total_pref_cost
                    Copy_List_Items(P.primary_next_hops,
                                    adv_node.primary_next_hops)
                elif total_pref_cost == min_total_pref_cost:
                    for nh_intf in adv_node.primary_next_hops:
                        Add_Item_To_List_If_New(P.primary_next_hops,
                                                nh_intf)

    def Select_Alts_For_One_Src_To_Named_Proxy_Nodes(topo,src):
        for prefix in topo.named_proxy_dict:
            P = topo.named_proxy_dict[prefix]
            P.alt_list = []
            for failed_intf in P.primary_next_hops:
                alt = Alternate()
                alt.failed_intf = failed_intf
                if failed_intf not in src.island_intf_list:
                    alt.info = 'PRIM_NH_FOR_PROXY_NODE_NOT_IN_ISLAND'
                elif P.pnar1 is None:
                    alt.info = 'NO_PNARs_EXIST_FOR_THIS_PREFIX'
                elif src is P.pnar1.node:
                    alt.info = 'SRC_IS_PNAR'
                elif P.pnar2 is not None and src is P.pnar2.node:
                    alt.info = 'SRC_IS_PNAR'
                elif P.pnar2 is None:
                    #inherit alternates from the only pnar.
                    alt.info = Select_Alternates(P.pnar1.node,
                                failed_intf.remote_node, failed_intf)
                elif failed_intf in src.island_intf_list:
                    alt.info = Select_Alternates_Proxy_Node(P,
                                failed_intf.remote_node, failed_intf)

                if alt.info == 'USE_RED_OR_BLUE':
                    alt.red_or_blue = \
                        random.choice(['USE_RED','USE_BLUE'])
                if (alt.info == 'USE_BLUE'
                    or alt.red_or_blue == 'USE_BLUE'):
                    Copy_List_Items(alt.nh_list, P.blue_next_hops)
                    alt.fec = 'BLUE'
                    alt.prot = 'NODE_PROTECTION'
                elif (alt.info == 'USE_RED'
                      or alt.red_or_blue == 'USE_RED'):
                    Copy_List_Items(alt.nh_list, P.red_next_hops)
                    alt.fec = 'RED'
                    alt.prot = 'NODE_PROTECTION'
                elif (alt.info == 'PRIM_NH_IS_D_OR_OP_FOR_D'
                    or alt.info == 'PRIM_NH_IS_OP_FOR_BOTH_X_AND_Y'):
                    if failed_intf.OUTGOING and failed_intf.INCOMING:
                        # cut-link: if there are parallel cut links, use
                        # the link(s) with lowest metric that are not
                        # primary intf or None
                        cand_alt_list = [None]
                        min_metric = 2147483647
                        for intf in src.island_intf_list:
                            if ( intf is not failed_intf and
                                 (intf.remote_node is
                                 failed_intf.remote_node)):
                                if intf.metric < min_metric:
                                    cand_alt_list = [intf]
                                    min_metric = intf.metric
                                elif intf.metric == min_metric:
                                    cand_alt_list.append(intf)
                        if cand_alt_list != [None]:
                            alt.fec = 'GREEN'
                            alt.prot = 'PARALLEL_CUTLINK'
                        else:
                            alt.fec = 'NO_ALTERNATE'
                            alt.prot = 'NO_PROTECTION'
                        Copy_List_Items(alt.nh_list, cand_alt_list)
                    else:
                        # set Z as the node to inherit blue next hops from
                        if alt.info == 'PRIM_NH_IS_D_OR_OP_FOR_D':
                            Z = P.pnar1.node
                        else:
                            Z = P
                        if failed_intf in Z.red_next_hops:
                            Copy_List_Items(alt.nh_list, Z.blue_next_hops)
                            alt.fec = 'BLUE'
                            alt.prot = 'LINK_PROTECTION'
                        else:
                            assert(failed_intf in Z.blue_next_hops)
                            Copy_List_Items(alt.nh_list, Z.red_next_hops)
                            alt.fec = 'RED'
                            alt.prot = 'LINK_PROTECTION'

                elif alt.info == 'PRIM_NH_FOR_PROXY_NODE_NOT_IN_ISLAND':
                    if (P.pnar2 == None and src is P.pnar1.node):
                        #MRT Island is singly connected to non-island dest
                        alt.fec = 'NO_ALTERNATE'
                        alt.prot = 'NO_PROTECTION'
                    elif P.node_id in src.blue_to_green_nh_dict:
                        # blue to P goes to failed LFIN so use red to P
                        Copy_List_Items(alt.nh_list, P.red_next_hops)
                        alt.fec = 'RED'
                        alt.prot = 'LINK_PROTECTION'
                    elif P.node_id in src.red_to_green_nh_dict:
                        # red to P goes to failed LFIN so use blue to P
                        Copy_List_Items(alt.nh_list, P.blue_next_hops)
                        alt.fec = 'BLUE'
                        alt.prot = 'LINK_PROTECTION'
                    else:
                        Copy_List_Items(alt.nh_list, P.blue_next_hops)
                        alt.fec = 'BLUE'
                        alt.prot = 'LINK_PROTECTION'
                elif alt.info == 'TEMP_NO_ALTERNATE':
                    alt.fec = 'NO_ALTERNATE'
                    alt.prot = 'NO_PROTECTION'

                P.alt_list.append(alt)

    def Run_Basic_MRT_for_One_Source(topo, src):
        MRT_Island_Identification(topo, src, 0, 0)
        Set_Island_Intf_and_Node_Lists(topo)
        Set_GADAG_Root(topo,src)
        Sort_Interfaces(topo)
        Run_Lowpoint(topo)
        Assign_Remaining_Lowpoint_Parents(topo)
        Construct_GADAG_via_Lowpoint(topo)
        Run_Assign_Block_ID(topo)
        Add_Undirected_Links(topo)
        Compute_MRT_NH_For_One_Src_To_Island_Dests(topo,src)
        Store_MRT_Nexthops_For_One_Src_To_Island_Dests(topo,src)
        Select_Alts_For_One_Src_To_Island_Dests(topo,src)
        Store_Primary_and_Alts_For_One_Src_To_Island_Dests(topo,src)

    def Store_GADAG_and_Named_Proxies_Once(topo):
        for node in topo.node_list:
            for intf in node.intf_list:
                if intf.OUTGOING:
                    intf.SIMULATION_OUTGOING = True
                else:
                    intf.SIMULATION_OUTGOING = False
        for prefix in topo.named_proxy_dict:
            P = topo.named_proxy_dict[prefix]
            topo.stored_named_proxy_dict[prefix] = P

    def Run_Basic_MRT_for_All_Sources(topo):
        for src in topo.node_list:
            Reset_Computed_Node_and_Intf_Values(topo)
            Run_Basic_MRT_for_One_Source(topo,src)
            if src is topo.gadag_root:
                Store_GADAG_and_Named_Proxies_Once(topo)
```

```
def Run_MRT_for_One_Source(topo, src):
    MRT_Island_Identification(topo, src, 0, 0)
    Set_Island_Intf_and_Node_Lists(topo)
    Set_GADAG_Root(topo,src)
    Sort_Interfaces(topo)
    Run_Lowpoint(topo)
    Assign_Remaining_Lowpoint_Parents(topo)
    Construct_GADAG_via_Lowpoint(topo)
    Run_Assign_Block_ID(topo)
    Add_Undirected_Links(topo)
    Compute_MRT_NH_For_One_Src_To_Island_Dests(topo,src)
    Store_MRT_Nexthops_For_One_Src_To_Island_Dests(topo,src)
    Select_Alts_For_One_Src_To_Island_Dests(topo,src)
    Store_Primary_and_Alts_For_One_Src_To_Island_Dests(topo,src)
    Create_Basic_Named_Proxy_Nodes(topo)
    Attach_Named_Proxy_Nodes(topo)
    Compute_MRT_NHs_For_One_Src_To_Named_Proxy_Nodes(topo,src)
    Store_MRT_NHs_For_One_Src_To_Named_Proxy_Nodes(topo,src)
    Compute_Primary_NHs_For_One_Src_To_Named_Proxy_Nodes(topo,src)
    Store_Primary_NHs_For_One_Src_To_Named_Proxy_Nodes(topo,src)
    Select_Alts_For_One_Src_To_Named_Proxy_Nodes(topo,src)
    Store_Alts_For_One_Src_To_Named_Proxy_Nodes(topo,src)

def Run_Prim_SPF_for_One_Source(topo,src):
    Normal_SPF(topo, src)
    Store_Primary_NHs_For_One_Source_To_Nodes(topo,src)
    Create_Basic_Named_Proxy_Nodes(topo)
    Compute_Primary_NHs_For_One_Src_To_Named_Proxy_Nodes(topo,src)
    Store_Primary_NHs_For_One_Src_To_Named_Proxy_Nodes(topo,src)

def Run_MRT_for_All_Sources(topo):
    for src in topo.node_list:
        Reset_Computed_Node_and_Intf_Values(topo)
        if src in topo.island_node_list_for_test_gr:
            # src runs MRT if it is in same MRT island as test_gr
            Run_MRT_for_One_Source(topo,src)
            if src is topo.gadag_root:
                Store_GADAG_and_Named_Proxies_Once(topo)
        else:
            # src still runs SPF if not in MRT island
            Run_Prim_SPF_for_One_Source(topo,src)

def Write_Output_To_Files(topo,file_prefix):
    Write_GADAG_To_File(topo,file_prefix)
    Write_Both_MRTs_For_All_Dests_To_File(topo,file_prefix)
    Write_Alternates_For_All_Dests_To_File(topo,file_prefix)
```

```
def Create_Basic_Topology_Input_File(filename):
    data = [[01,02,10],[02,03,10],[03,04,11],[04,05,10,20],[05,06,10],
            [06,07,10],[06,07,10],[06,07,15],[07,01,10],[07,51,10],
            [51,52,10],[52,53,10],[53,03,10],[01,55,10],[55,06,10],
            [04,12,10],[12,13,10],[13,14,10],[14,15,10],[15,16,10],
            [16,17,10],[17,04,10],[05,76,10],[76,77,10],[77,78,10],
            [78,79,10],[79,77,10]]
    with open(filename + '.csv', 'w') as topo_file:
        for item in data:
            if len(item) > 3:
                line = (str(item[0])+','+str(item[1])+','+
                        str(item[2])+','+str(item[3])+'\n')
            else:
                line = (str(item[0])+','+str(item[1])+','+
                        str(item[2])+'\n')
            topo_file.write(line)

def Create_Complex_Topology_Input_File(filename):
    data = [[01,02,10],[02,03,10],[03,04,11],[04,05,10,20],[05,06,10],
            [06,07,10],[06,07,10],[06,07,15],[07,01,10],[07,51,10],
            [51,52,10],[52,53,10],[53,03,10],[01,55,10],[55,06,10],
            [04,12,10],[12,13,10],[13,14,10],[14,15,10],[15,16,10],
            [16,17,10],[17,04,10],[05,76,10],[76,77,10],[77,78,10],
            [78,79,10],[79,77,10]]
    with open(filename + '.csv', 'w') as topo_file:
        for item in data:
            if len(item) > 3:
                line = (str(item[0])+','+str(item[1])+','+
                        str(item[2])+','+str(item[3])+'\n')
            else:
                line = (str(item[0])+','+str(item[1])+','+
                        str(item[2])+'\n')
            topo_file.write(line)

    data = [[01,0],[02,0],[03,0],[04,0],[05,0],
            [06,0],[07,0],
            [51,0],[55,0],
            [12,0],[13,0],[14,0],[15,0],
            [16,0],[17,0],[76,0],[77,0],
            [78,0],[79,0]]
    with open(filename + '.profile', 'w') as topo_file:
        for item in data:
            line = (str(item[0])+','+str(item[1])+'\n')
            topo_file.write(line)

    data = [[2001,05,100],[2001,07,120],[2001,03,130],
            [2002,13,100],[2002,15,110],
            [2003,52,100],[2003,78,100]]
    with open(filename + '.prefix', 'w') as topo_file:
        for item in data:
            line = (str(item[0])+','+str(item[1])+','+
                    str(item[2])+'\n')
            topo_file.write(line)

def Generate_Basic_Topology_and_Run_MRT():
    this_gadag_root = 3
    Create_Basic_Topology_Input_File('basic_topo_input')
    topo = Create_Topology_From_File('basic_topo_input')
    res_file_base = 'basic_topo'
    Compute_Island_Node_List_For_Test_GR(topo, this_gadag_root)
    Raise_GADAG_Root_Selection_Priority(topo,this_gadag_root)
    Run_Basic_MRT_for_All_Sources(topo)
    Write_Output_To_Files(topo, res_file_base)

def Generate_Complex_Topology_and_Run_MRT():
    this_gadag_root = 3
    Create_Complex_Topology_Input_File('complex_topo_input')
    topo = Create_Topology_From_File('complex_topo_input')
    Add_Profile_IDs_from_File(topo,'complex_topo_input')
    Add_Prefix_Advertisements_From_File(topo,'complex_topo_input')
    Compute_Island_Node_List_For_Test_GR(topo, this_gadag_root)
    Add_Prefixes_for_Non_Island_Nodes(topo)
    res_file_base = 'complex_topo'
    Raise_GADAG_Root_Selection_Priority(topo,this_gadag_root)
    Run_MRT_for_All_Sources(topo)
    Write_Output_To_Files(topo, res_file_base)

Generate_Basic_Topology_and_Run_MRT()

Generate_Complex_Topology_and_Run_MRT()

<CODE ENDS>
```

Appendix B. Constructing a GADAG Using SPFs

The basic idea in this method for constructing a GADAG is to use slightly modified SPF computations to find ears. In every block, an SPF computation is first done to find a cycle from the local root and then SPF computations in that block find ears until there are no more interfaces to be explored. The used result from the SPF computation is the path of interfaces indicated by following the previous hops from the minimized IN_GADAG node back to the SPF root.

To do this, first all cut-vertices must be identified and localroots assigned as specified in Figure 12.

The slight modifications to the SPF are as follows. The root of the block is referred to as the block-root; it is either the GADAG root or a cut-vertex.

a. The SPF is rooted at a neighbor x of an IN_GADAG node y. All links between y and x are marked as TEMP_UNUSABLE. They should not be used during the SPF computation.

b. If y is not the block-root, then it is marked TEMP_UNUSABLE. It should not be used during the SPF computation. This prevents ears from starting and ending at the same node and avoids cycles; the exception is because cycles to/from the block-root are acceptable and expected.

c. Do not explore links to nodes whose localroot is not the block-root. This keeps the SPF confined to the particular block.

d. Terminate when the first IN_GADAG node z is minimized.

e. Respect the existing directions (e.g., INCOMING, OUTGOING, UNDIRECTED) already specified for each interface.

```
 Mod_SPF(spf_root, block_root)
    Initialize spf_heap to empty
    Initialize nodes' spf_metric to infinity
    spf_root.spf_metric = 0
    insert(spf_heap, spf_root)
    found_in_gadag = false
    while (spf_heap is not empty) and (found_in_gadag is false)
        min_node = remove_lowest(spf_heap)
        if min_node.IN_GADAG
           found_in_gadag = true
        else
           foreach interface intf of min_node
              if ((intf.OUTGOING or intf.UNDIRECTED) and
                  ((intf.remote_node.localroot is block_root) or
                   (intf.remote_node is block_root)) and
                  (intf.remote_node is not TEMP_UNUSABLE) and
                  (intf is not TEMP_UNUSABLE))
                 path_metric = min_node.spf_metric + intf.metric
                 if path_metric < intf.remote_node.spf_metric
                    intf.remote_node.spf_metric = path_metric
                    intf.remote_node.spf_prev_intf = intf
                    insert_or_update(spf_heap, intf.remote_node)
    return min_node
 SPF_for_Ear(cand_intf.local_node,cand_intf.remote_node, block_root,
             method)
    Mark all interfaces between cand_intf.remote_node
               and cand_intf.local_node as TEMP_UNUSABLE
    if cand_intf.local_node is not block_root
       Mark cand_intf.local_node as TEMP_UNUSABLE
    Initialize ear_list to empty
    end_ear = Mod_SPF(spf_root, block_root)
    y = end_ear.spf_prev_hop
    while y.local_node is not spf_root
      add_to_list_start(ear_list, y)
      y.local_node.IN_GADAG = true
      y = y.local_node.spf_prev_intf
    if(method is not hybrid)
       Set_Ear_Direction(ear_list, cand_intf.local_node,
                         end_ear,block_root)
    Clear TEMP_UNUSABLE from all interfaces between
          cand_intf.remote_node and cand_intf.local_node
    Clear TEMP_UNUSABLE from cand_intf.local_node
 return end_ear
```

Figure 31: Modified SPF for GADAG Construction

Assume that an ear is found by going from y to x and then running an SPF that terminates by minimizing z (e.g., y<->x...q<->z). Now it is necessary to determine the direction of the ear; if y<<z, then the path should be y->x...q->z; but if y>>z, then the path should be y<-x...q<-z. In Section 5.5, the same problem was handled by finding all ears that started at a node before looking at ears starting at nodes higher in the partial order. In this GADAG construction method, using that approach could mean that new ears aren't added in order of their total cost since all ears connected to a node would need to be found before additional nodes could be found.

The alternative is to track the order relationship of each node with respect to every other node. This can be accomplished by maintaining two sets of nodes at each node. The first set, Higher_Nodes, contains all nodes that are known to be ordered above the node. The second set, Lower_Nodes, contains all nodes that are known to be ordered below the node. This is the approach used in this GADAG construction method.

```
      Set_Ear_Direction(ear_list, end_a, end_b, block_root)
        // Default of A_TO_B for the following cases:
        // (a) end_a and end_b are the same (root)
        // or (b) end_a is in end_b's Lower Nodes
        // or (c) end_a and end_b were unordered with respect to each
        //        other
        direction = A_TO_B
        if (end_b is block_root) and (end_a is not end_b)
           direction = B_TO_A
        else if end_a is in end_b.Higher_Nodes
           direction = B_TO_A
        if direction is B_TO_A
           foreach interface i in ear_list
               i.UNDIRECTED = false
               i.INCOMING = true
               i.remote_intf.UNDIRECTED = false
               i.remote_intf.OUTGOING = true
        else
           foreach interface i in ear_list
               i.UNDIRECTED = false
               i.OUTGOING = true
               i.remote_intf.UNDIRECTED = false
               i.remote_intf.INCOMING = true
         if end_a is end_b
            return
         // Next, update all nodes' Lower_Nodes and Higher_Nodes
         if (end_a is in end_b.Higher_Nodes)
            foreach node x where x.localroot is block_root
                if end_a is in x.Lower_Nodes
                   foreach interface i in ear_list
                      add i.remote_node to x.Lower_Nodes
                if end_b is in x.Higher_Nodes
                   foreach interface i in ear_list
                      add i.local_node to x.Higher_Nodes
          else
            foreach node x where x.localroot is block_root
                if end_b is in x.Lower_Nodes
                   foreach interface i in ear_list
                      add i.local_node to x.Lower_Nodes
                if end_a is in x.Higher_Nodes
                   foreach interface i in ear_list
                      add i.remote_node to x.Higher_Nodes
```

Figure 32: Algorithm to Assign Links of an Ear Direction

A goal of this GADAG construction method is to find the shortest cycles and ears. An ear is started by going to a neighbor x of an IN_GADAG node y. The path from x to an IN_GADAG node is minimal, since it is computed via SPF. Since a shortest path is made of shortest paths, to find the shortest ears requires reaching from the set of IN_GADAG nodes to the closest node that isn't IN_GADAG. Therefore, an ordered tree is maintained of interfaces that could be explored from the IN_GADAG nodes. The interfaces are ordered by their characteristics of metric, local loopback address, remote loopback address, and ifindex, based on the Interface_Compare function defined in Figure 14.

This GADAG construction method ignores interfaces picked from the ordered list that belong to the block root if the block in which the interface is present already has an ear that has been computed. This is necessary since we allow at most one incoming interface to a block root in each block. This requirement stems from the way next hops are computed as was seen in Section 5.7. After any ear gets computed, we traverse the newly added nodes to the GADAG and insert interfaces whose far end is not yet on the GADAG to the ordered tree for later processing.

Finally, cut-links are a special case because there is no point in doing an SPF on a block of two nodes. The algorithm identifies cut-links simply as links where both ends of the link are cut-vertices. Cut-links can simply be added to the GADAG with both OUTGOING and INCOMING specified on their interfaces.

```
  add_eligible_interfaces_of_node(ordered_intfs_tree,node)
     for each interface of node
        if intf.remote_node.IN_GADAG is false
           insert(intf,ordered_intfs_tree)

  check_if_block_has_ear(x,block_id)
     block_has_ear = false
        for all interfaces of x
           if ( (intf.remote_node.block_id == block_id) &&
                 intf.remote_node.IN_GADAG )
              block_has_ear = true
  return block_has_ear

  Construct_GADAG_via_SPF(topology, root)
    Compute_Localroot (root,root)
    Assign_Block_ID(root,0)
    root.IN_GADAG = true
       add_eligible_interfaces_of_node(ordered_intfs_tree,root)
    while ordered_intfs_tree is not empty
       cand_intf = remove_lowest(ordered_intfs_tree)
       if cand_intf.remote_node.IN_GADAG is false
          if L(cand_intf.remote_node) == D(cand_intf.remote_node)
             // Special case for cut-links
```

```
                cand_intf.UNDIRECTED = false
                cand_intf.remote_intf.UNDIRECTED = false
                cand_intf.OUTGOING = true
                cand_intf.INCOMING = true
                cand_intf.remote_intf.OUTGOING = true
                cand_intf.remote_intf.INCOMING = true
                cand_intf.remote_node.IN_GADAG = true
             add_eligible_interfaces_of_node(
                         ordered_intfs_tree,cand_intf.remote_node)
          else
             if (cand_intf.remote_node.local_root ==
                 cand_intf.local_node) &&
                 check_if_block_has_ear(cand_intf.local_node,
                              cand_intf.remote_node.block_id))
                 /* Skip the interface since the block root
                 already has an incoming interface in the
                 block */
             else
             ear_end = SPF_for_Ear(cand_intf.local_node,
                     cand_intf.remote_node,
                     cand_intf.remote_node.localroot,
                     SPF method)
             y = ear_end.spf_prev_hop
             while y.local_node is not cand_intf.local_node
                 add_eligible_interfaces_of_node(
                     ordered_intfs_tree, y.local_node)
                 y = y.local_node.spf_prev_intf
```

Figure 33: SPF-Based Method for GADAG Construction

Appendix C. Constructing a GADAG Using a Hybrid Method

The idea of this method is to combine the salient features of the lowpoint inheritance and SPF methods. To this end, we process nodes as they get added to the GADAG just like in the lowpoint inheritance by maintaining a stack of nodes. This ensures that we do not need to maintain lower and higher sets at each node to ascertain ear directions since the ears will always be directed from the node being processed towards the end of the ear. To compute the ear however, we resort to an SPF to have the possibility of better ears (path lengths) thus giving more flexibility than the restricted use of lowpoint/dfs parents.

Regarding ears involving a block root, unlike the SPF method, which ignored interfaces of the block root after the first ear, in the hybrid method we would have to process all interfaces of the block root before moving on to other nodes in the block since the direction of an ear is predetermined. Thus, whenever the block already has an ear computed, and we are processing an interface of the block root, we mark the block root as unusable before the SPF run that computes the ear. This ensures that the SPF terminates at some node other than the block-root. This in turn guarantees that the block-root has only one incoming interface in each block, which is necessary for correctly computing the next hops on the GADAG.

As in the SPF GADAG, bridge ears are handled as a special case.

The entire algorithm is shown below in Figure 34.

```
find_spf_stack_ear(stack, x, y, xy_intf, block_root)
   if L(y) == D(y)
      // Special case for cut-links
      xy_intf.UNDIRECTED = false
      xy_intf.remote_intf.UNDIRECTED = false
      xy_intf.OUTGOING = true
      xy_intf.INCOMING = true
      xy_intf.remote_intf.OUTGOING = true
      xy_intf.remote_intf.INCOMING = true
      xy_intf.remote_node.IN_GADAG = true
      push y onto stack
      return
   else
      if (y.local_root == x) &&
           check_if_block_has_ear(x,y.block_id)
         //Avoid the block root during the SPF
         Mark x as TEMP_UNUSABLE
      end_ear = SPF_for_Ear(x,y,block_root,hybrid)
      If x was set as TEMP_UNUSABLE, clear it
      cur = end_ear
      while (cur != y)
         intf = cur.spf_prev_hop
         prev = intf.local_node
         intf.UNDIRECTED = false
         intf.remote_intf.UNDIRECTED = false
         intf.OUTGOING = true
         intf.remote_intf.INCOMING = true
         push prev onto stack
      cur = prev
      xy_intf.UNDIRECTED = false
      xy_intf.remote_intf.UNDIRECTED = false
      xy_intf.OUTGOING = true
      xy_intf.remote_intf.INCOMING = true
      return
Construct_GADAG_via_hybrid(topology,root)
   Compute_Localroot (root,root)
   Assign_Block_ID(root,0)
   root.IN_GADAG = true
   Initialize Stack to empty
   push root onto Stack
   while (Stack is not empty)
      x = pop(Stack)
      for each interface intf of x
         y = intf.remote_node
         if y.IN_GADAG is false
            find_spf_stack_ear(stack, x, y, intf, y.block_root)
```

Figure 34: Hybrid GADAG Construction Method

Acknowledgements

The authors would like to thank Shraddha Hegde, Eric Wu, Janos Farkas, Stewart Bryant, Alvaro Retana, and Deccan (Shaofu Peng) for their suggestions and review. We would also like to thank Anil Kumar SN for his assistance in clarifying the algorithm description and pseudocode.

Authors' Addresses

Gabor Sandor Enyedi
Ericsson
Konyves Kalman krt 11
Budapest 1097
Hungary

Email: Gabor.Sandor.Enyedi@ericsson.com

Andras Csaszar
Ericsson
Konyves Kalman krt 11
Budapest 1097
Hungary

Email: Andras.Csaszar@ericsson.com

Alia Atlas
Juniper Networks
10 Technology Park Drive
Westford, MA 01886
United States

Email: akatlas@juniper.net

Chris Bowers
Juniper Networks
1194 N. Mathilda Ave.
Sunnyvale, CA 94089
United States

Email: cbowers@juniper.net

Abishek Gopalan
University of Arizona
1230 E Speedway Blvd.
Tucson, AZ 85721
United States

Email: abishek@ece.arizona.edu