Routing Area Working Group
Internet-Draft
Intended status: Informational
Expires: January 16, 2014

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

July 15, 2013

Algorithms for computing Maximally Redundant Trees for IP/LDP Fast-Reroute
draft-enyedi-rtgwg-mrt-frr-algorithm-03

Abstract

A complete solution for IP and LDP Fast-Reroute using Maximally Redundant Trees is presented in [I-D.ietf-rtgwg-mrt-frr-architecture]. This document defines 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.

Table of Contents

1. Introduction

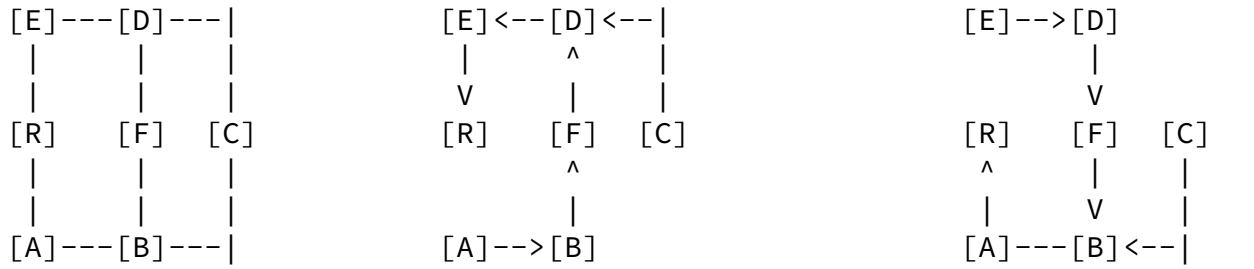
MRT Fast-Reroute requires that packets can be forwarded not only on the shortest-path tree, but also on two Maximally Redundant Trees (MRTs), referred to as the MRT-Blue and the MRT-Red. A router which experiences a local failure must also have pre-determined which alternate to use. This document defines 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].

Just as packets routed on a hop-by-hop basis require that each router compute a shortest-path tree which 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.

As now, a router's FIB will 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. LFAs that are downstream paths, may be prefered when available and that policy-based alternate selection process[I-D.ietf-rtgwg-lfa-manageability] is not captured in this document.



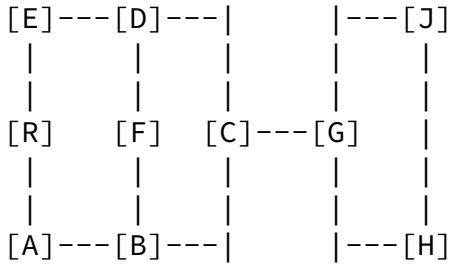
(a) a 2-connected graph

(b) MRT-Blue towards R

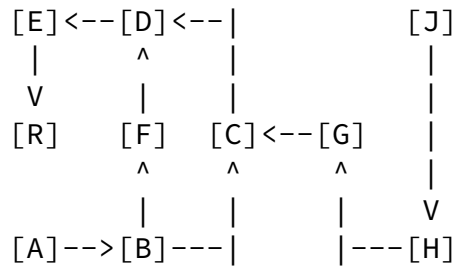
(c) MRT-Red towards R

Figure 1

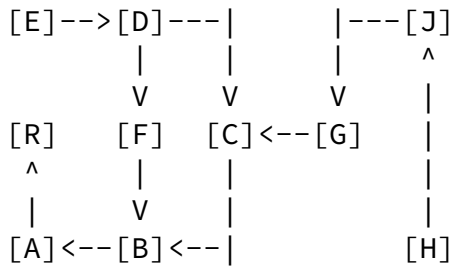
Algorithms for computing MRTs can handle arbitrary network topologies where the whole network graph is not 2-connected, as in Figure 2, as well as the easier case where the network graph is 2-connected (Figure 1). Each MRT is a spanning tree. The pair of MRTs provide two paths from every node X to the 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.



(a) a graph that isn't 2-connected



(b) MRT-Blue towards R



(c) MRT-Red towards R

Figure 2

2. Terminology and Definitions

network graph: A graph that reflects the network topology where all links connect exactly two nodes and broadcast links have been transformed into the standard pseudo-node representation.

Redundant Trees (RT): A pair of trees where the path from any node X to the root R on the first tree is node-disjoint with the path from the same node X to the root along the second tree. These can be computed in 2-connected graphs.

Maximally Redundant Trees (MRT): A pair of trees where the path from any node X to the root R along the first tree and the path from the same node X to the root along the second tree 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. Any RT is an MRT but many MRTs are not RTs.

MRT-Red: MRT-Red is used to describe one of the two MRTs; it is used to described the associated forwarding topology and MT-ID. Specifically, MRT-Red is the decreasing MRT where links in the GADAG are taken in the direction from a higher topologically ordered node to a lower one.

MRT-Blue: MRT-Blue is used to describe one of the two MRTs; it is used to described the associated forwarding topology and MT-ID. Specifically, MRT-Blue is the increasing MRT where links in the GADAG are taken in the direction from a lower topologically ordered node to a higher one.

cut-vertex: A vertex whose removal partitions the network.

cut-link: A link whose removal partitions the network. A cut-link by definition must be connected between two cut-vertices. If there are multiple parallel links, then they are referred to as cut-links in this document if removing the set of parallel links would partition the network.

2-connected: A graph that has no cut-vertices. This is a graph that requires two nodes to be removed before the network is

partitioned.

spanning tree: A tree containing links 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.

2-connected cluster: A maximal set of nodes that are 2-connected. In a network graph with at least one cut-vertex, there will be multiple 2-connected clusters.

block: Either a 2-connected cluster, a cut-edge, or an isolated vertex.

DAG: Directed Acyclic Graph - a digraph containing no directed cycle.

ADAG: Almost Directed Acyclic Graph - a digraph that can be transformed into a DAG whith removing a single node (the root node).

GADAG: Generalized ADAG - a digraph, which has only ADAGs as all of its blocks. The root of such a block is the node closest to the global root (e.g. with uniform link costs).

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 not-yet-included-in-the-GADAG nodes 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.

proxy-node: A node added to the network graph to represent a multi-homed prefix or routers outside the local MRT-fast-reroute-supporting island of routers. The key property of proxy-nodes is that traffic cannot transit them.

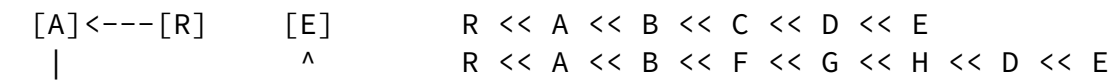
3. Algorithm Key Concepts

There are five key concepts that are critical for understanding the MRT Lowpoint algorithm and other algorithms for computing MRTs. 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 Low-Point 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-edge or an isolated node. The fifth is the idea of a local-root for each node; this is used to compute ADAGs in each block.

3.1. 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.



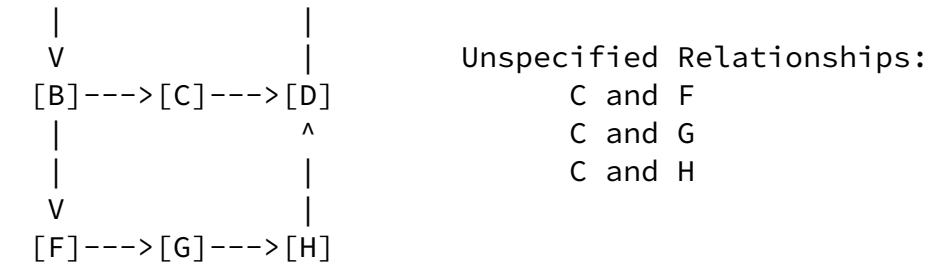


Figure 3: Directed Graph showing a Partial Order

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.

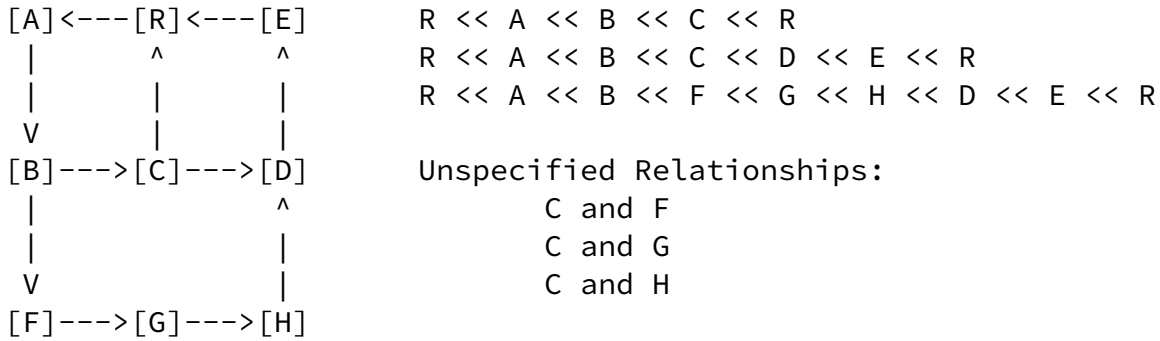


Figure 4: ADAG showing a Partial Order with R lowest and highest

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 inceasing path.

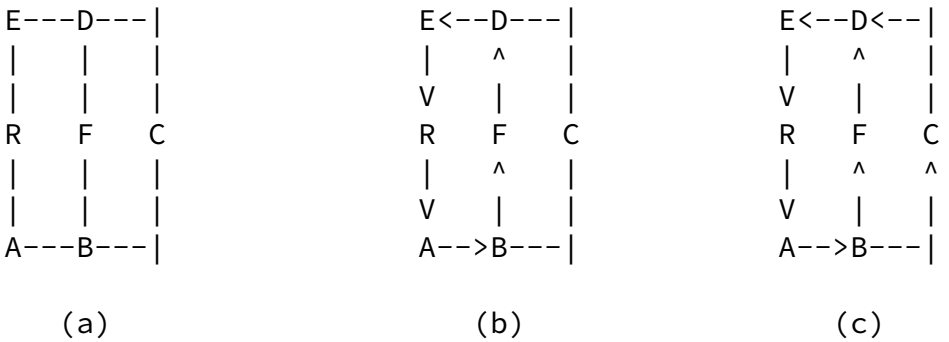
When following the increasing paths, it is possible to pick multiple higher nodes and still have the certainty that those paths will be disjoint from the decreasing paths. E.g. in the previous example node B has multiple possibilities to forward packets along an increasing path: it can either forward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

3.2. 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 4.4.

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.



(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 sequel), 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 were strictly greater than 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 already 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 4.6

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, we can always add ears to it: just select a non-ready neighbor N of a ready node Q, such that Q is not the root, find a path from N to the root 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 second 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 exists 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

Algorithm Figure 6 merely requires that a cycle or ear be selected without specifying how. Regardless of the way of 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's method using Low-Point Inheritance is defined in Section 4.4. Other methods are described in the Appendices (Appendix A and Appendix B).

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.

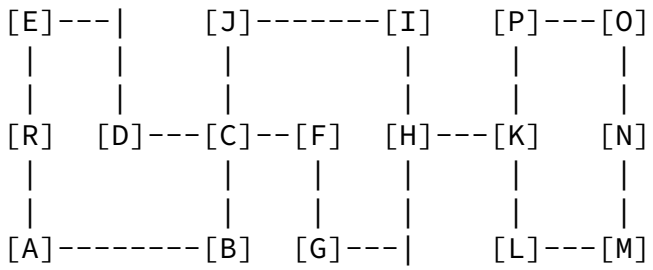
3.3. Low-Point Values and Their Uses

A basic way of computing a spanning tree on a network graph is to run a depth-first-search, 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.

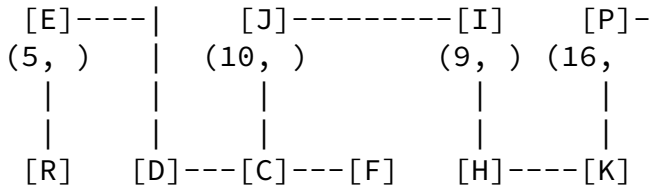
```
global_variable: dfs_number

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 root)
   dfs_number = 0
   DFS_Visit(root, NONE)
```

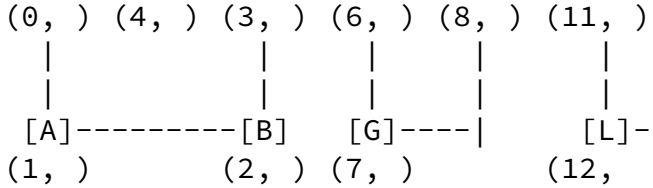
Figure 7: Basic Depth-First Search algorithm

Given a node x, one can compute the minimal DFS number of the neighbours of x, i.e. min(D(w) if (x,w) is a link). This gives the highest attachment point neighbouring x. What is interesting, though, is what is the highest attachment point from x and x's descendants. This is what is determined by computing the Low-Point value, as given in Algorithm Figure 9 and illustrated on a graph in Figure 8.

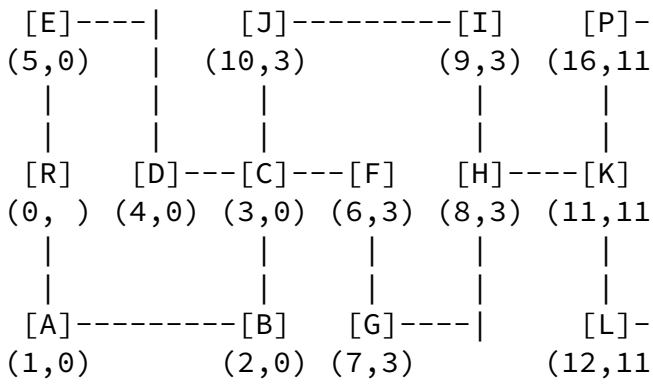


(a) a non-2-connected graph





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



(c) with low-point values assigned (D(x), L(x))

Figure 8

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
   x.lowpoint_parent = NONE
   for each 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)
         x.lowpoint_parent = intf.remote_node
         x.lowpoint_parent_intf = intf

Run_Lowpoint(node root)
   dfs_number = 0
```



```
Lowpoint_Visit(root, NONE, NONE)
```

Figure 9: Computing Low-Point value

From the low-point value and lowpoint parent, there are two very useful things which motivate our computation.

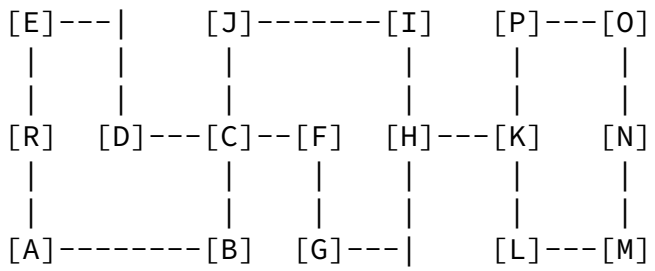
First, if there is a child c of x such that L(c) >= D(x), then there are no paths in the network graph that go from c or its descendants to an ancestor of x - and therefore x is a cut-vertex. This is useful because it allows identification of the cut-vertices and thus the blocks. As seen in Figure 8, 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.

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

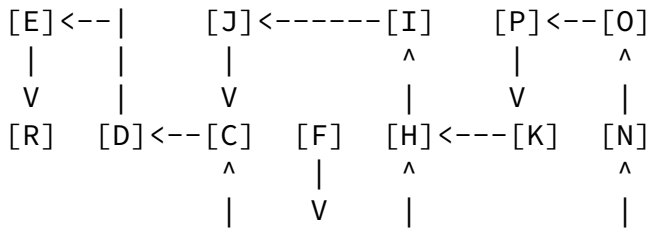
link [x, x.dfs_parent] in either direction. The full path need not be taken, but this gives a way of finding an initial cycle and then ears.

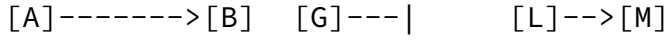
3.4. Blocks in a Graph

A key idea for an MRT 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.

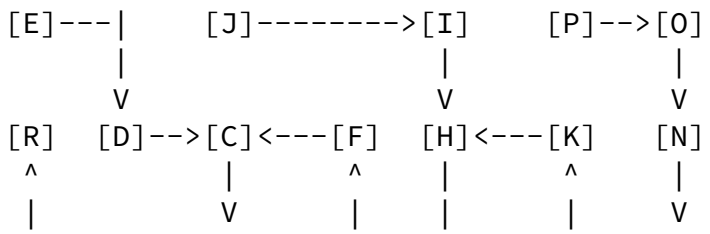


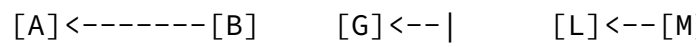
(a) A graph with four blocks that are:
3 2-connected clusters and a cut-link





(b) MRT-Blue





(c) MRT-Red

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-edge 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-edge that is block 4. No two blocks can have more than one common node, since two blocks with at least 2 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 one in block 2 with C as a root, and finally, we need the last one 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-edge (as in block 4), that cut-edge 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 local-root to use for each block.

3.5. Determining Local-Root and Assigning Block-ID

Each node in a network graph has a local-root, which is the cut-vertex (or root) in the same block that is closest to the root. The

local-root is used to determine whether two nodes share a common block.

```
Compute_Localroot(node x, node localroot)
    x.localroot = localroot
    for each DFS child c
        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(root, root)
```

Figure 11: A method for computing local-roots

There are two different ways of computing the local-root for each node. The stand-alone method is given in Figure 11 and better illustrates the concept; it is used by the MRT algorithms given in the Appendices Appendix A and Appendix B. The method for local-root computation is used in the MRT Lowpoint algorithm for computing a GADAG using Low-Point inheritance and the essence of it is given in Figure 12.

```
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 local-roots

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

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

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

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

Once we have computed the local-root 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(root, max_block_id)
```

Figure 13: Assigning block id to identify blocks

4. Algorithm Sections

This 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. These work on a network graph after, for instance, its interfaces are ordered as per Figure 14.

1. Compute the local MRT Island for the particular MRT Profile. [See Section 4.1.]

2. Select the root to use for the GADAG. [See Section 4.2.]

3. Initialize all interfaces to UNDIRECTED. [See Section 4.3.]

4. Compute the DFS value,e.g. D(x), and lowpoint value, L(x). [See Figure 9.]

5. Construct the GADAG. [See Section 4.4]

6. Assign directions to all interfaces that are still UNDIRECTED. [See Section 4.5.]

7. From the computing router x, compute the next-hops for the MRT-Blue and MRT-Red. [See Section 4.6.]

8. Identify alternates for each next-hop to each destination by determining which one of the blue MRT and the red MRT the computing router x should select. [See Section 4.7.]

To ensure consistency in computation, all routers MUST order interfaces identically. This is necessary for the DFS, where the selection order of the interfaces to explore results in different trees, and for computing the GADAG, where the selection order of the interfaces to use to form ears can result in different GADAGs. 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.loopback_addr < b.neighbor.loopback_addr
      return A_LESS_THAN_B
   if b.neighbor.loopback_addr < a.neighbor.loopback_addr
      return B_LESS_THAN_A
   // Same metric to same node, so the order doesn't matter anymore.
   // To have a unique, consistent total order,
   // tie-break based on, for example, the link's linkData as
   // distributed in an OSPF Router-LSA
   if a.link_data < b.link_data
      return A_LESS_THAN_B
   return B_LESS_THAN_A
```

Figure 14: Rules for ranking multiple interfaces. Order is from low to high.

4.1. 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), exploring only links that aren't MRT-ineligible.

```
   MRT_Island_Identification(topology, computing_rtr, profile_id)
     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 interface in next_rtr
              if interface is not MRT-ineligible
                 if ((interface.remote_node supports profile_id) and
                     (interface.remote_node.IN_MRT_ISLAND is FALSE))
                    interface.remote_node.IN_MRT_ISLAND = TRUE
                    add_to_tail(explore_list, interface.remote_node)
```

Figure 15: MRT Island Identification

4.2. Root Selection

In [I-D.atlas-ospf-mrt], a mechanism is given for routers to advertise the GADAG Root Selection Priority and consistently select a GADAG Root inside the local MRT Island. 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.

Off-line analysis that considers the centrality of a router may help determine how good a choice a particular router is for the role of GADAG root.

4.3. 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 as unusable, whether because of configuration, IGP flooding (e.g. MRT-ineligible links in [I-D.atlas-ospf-mrt]), overload, or due to a transient cause such as [RFC3137]. In the algorithm description, it is assumed that such links and nodes will not be explored or used and no more discussion is given of this restriction.

4.4. MRT Lowpoint Algorithm: Computing GADAG using lowpoint inheritance

As discussed in Section 3.2, it is necessary to find ears from a node x that is already in the GADAG (known as IN_GADAG). There are two methods to find ears; both are required. The first is by going to a not IN_GADAG DFS-child and then following the chain of low-point parents until an IN_GADAG node is found. The second is by going to a not IN_GADAG neighbor 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 is 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 Low-Point inheritance, it is necessary that every node have a low-point parent that is not itself. If a node is a cut-vertex, that may not yet be the case. Therefore, any nodes without a low-point parent will have their low-point parent set to their DFS parent and their low-point 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 local-root, as described in Figure 12. This is further elaborated as follows. The local-root 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 local-root 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 local-root would be the correct local-root for all newly added nodes.

The pseudo-code for the GADAG algorithm (assuming that the adjustment of lowpoint for cut-vertices has been made) is shown in Figure 16.

```
        Construct_Ear(x, Stack, intf, 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 type is CHILD
                   cur_intf = cur_node.lowpoint_parent_intf
                   cur_node = cur_node.lowpoint_parent
                else type must be NEIGHBOR
                   cur_intf = cur_node.dfs_parent_intf
                   cur_node = cur_node.dfs_parent
             else
                not_done = false

          if (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
             localroot = x
          else
             // Inherit local-root 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, root)
         root.IN_GADAG = true
         root.localroot = root
         Initialize Stack to empty
         push root onto Stack
         while (Stack is not empty)
```



```
           x = pop(Stack)
           foreach 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 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, root)
```

Figure 16: Low-point Inheritance GADAG algorithm

4.5. Augmenting the GADAG by directing all links

The GADAG, regardless of the algorithm 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. Finally, all UNDIRECTED links are assigned a direction based upon the partial ordering. Any UNDIRECTED links that connect to a root of a block from within that block are assigned a direction INCOMING to that root. The exact details of this whole process are captured in Figure 17


```
Set_Block_Root_Incoming_Links(topo, root, mark_or_clear)
   foreach node x in topo
      if node x is a cut-vertex or root
         foreach interface i of x
            if (i.remote_node.localroot is x)
```

```
                    if i.UNDIRECTED
                       i.OUTGOING = true
                       i.remote_intf.INCOMING = true
                       i.UNDIRECTED = false
                       i.remote_intf.UNDIRECTED = false
                    if i.INCOMING
                       if mark_or_clear is mark
                          if i.OUTGOING  // a cut-edge
                             i.STORE_INCOMING = true
                             i.INCOMING = false
                             i.remote_intf.STORE_OUTGOING = true
                             i.remote_intf.OUTGOING = false
                          i.TEMP_UNUSABLE = true
                          i.remote_intf.TEMP_UNUSABLE = true
                       else
                          i.TEMP_UNUSABLE = false
                          i.remote_intf.TEMP_UNUSABLE = false
                    if i.STORE_INCOMING and (mark_or_clear is clear)
                       i.INCOMING = true
                       i.STORE_INCOMING = false
                       i.remote_intf.OUTGOING = true
                       i.remote_intf.STORE_OUTGOING = false

      Run_Topological_Sort_GADAG(topo, root)
         Set_Block_Root_Incoming_Links(topo, root, MARK)
         foreach node x
           set x.unvisited to the count of x's incoming interfaces
              that aren't marked TEMP_UNUSABLE
         Initialize working_list to empty
         Initialize topo_order_list to empty
         add_to_list_end(working_list, root)
         while working_list is not empty
            y = remove_start_item_from_list(working_list)
            add_to_list_end(topo_order_list, y)
            foreach interface i of y
                if (i.OUTGOING) and (not i.TEMP_UNUSABLE)
                   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
        Set_Block_Root_Incoming_Links(topo, root, CLEAR)

    Add_Undirected_Links(topo, root)
        Run_Topological_Sort_GADAG(topo, root)
        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, root)
```

Figure 17: 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 and how the appropriate alternate next-hops are selected is given in Section 4.8.

4.6. Compute MRT next-hops

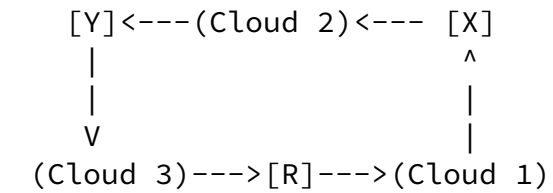
As was discussed in Section 3.1, once a ADAG is found, it is straightforward to find the next-hops from any node X to the ADAG root. However, in this algorithm, we want to reuse the common GADAG and find not only the one pair of MRTs rooted at the GADAG root with it, but find a pair rooted at each node. This is useful since it is significantly faster to compute. It may also provide easier troubleshooting of the MRT-Red and MRT-Blue.

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 two-connected. The generalization to multiple blocks is discussed in Section 4.6.4. The full algorithm is given in Section 4.6.5.

4.6.1. MRT next-hops to all nodes partially ordered with respect to the computing node

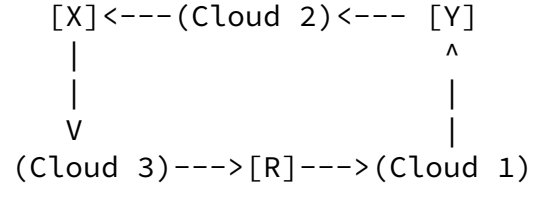
To find two node-disjoint paths from the computing router X to any node Y, depends upon whether Y >> X or Y << X. As shown in Figure 18, if Y >> X, then there is an increasing path that goes from X to Y without crossing R; this contains nodes in the interval [X,Y]. There is also a decreasing path that 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.



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

Figure 18: Y >> X

Similar logic applies if Y << X, as shown in Figure 19. 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].



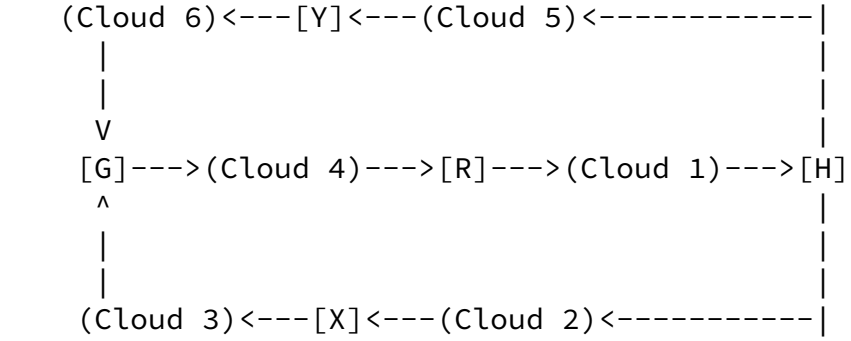
MRT-Blue path: X->Cloud 3->R->Cloud 1->Y

MRT-Red path: X->Cloud 2->Y

Figure 19: Y << X

4.6.2. MRT next-hops to all nodes not partially 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 20. 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.



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 20: 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 which 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 which is the largest one before R, so the MRT-Red path will never reach R; it will turn at M and decrease to Y.

4.6.3. 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 Section 4.6.1 and Section 4.6.2. Given these two ideas, how can we find the trees?

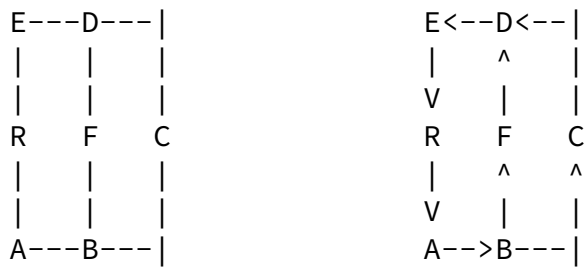
If some node X only wants to find the next-hops (which is usually the case for IP networks), it is enough to find which nodes are greater and less than X, and which are not ordered; this can be done by running an increasing-SPF and a decreasing-SPF rooted at X and not exploring any links from the ADAG root. (Traversal algorithms other than SPF could safely be used instead where one traversal takes the links in their given directions and the other reverses the links' directions.)

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. An 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, which were not reached, we have the next-hops as well. The increasing MRT-Blue next-hop for a node, which is not ordered, 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 identity of G and H is not determined.

The final case to considered is when the root R computes its own next-hops. Since the 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 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.



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

Figure 21

As an example consider the situation depicted in Figure 21. There node C runs an increasing-SPF and a decreasing-SPF The increasing-SPF reaches D, E and R and the decreasing-SPF reaches B, A and R. So towards E the increasing next-hop is D (it was reached though D), and the decreasing next-hop is B (since R was reached though B). Since both D and B, A and R will compute the next hops similarly, the packets will reach E.

We have 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 B is ordered with F, it will find, for its MRT-Blue, a real increasing next-hop, so packet forwarded to B will get to F on path C-B-F. Similarly, D will have, for its MRT-Red, a real decreasing next-hop, and the packet will use path C-D-F.

4.6.4. Generalizing for graph that isn't 2-connected

If a graph isn't 2-connected, then the basic approach given in Section 4.6.3 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 local-root should be used. This has the following implications:

    a. The links from X's local-root 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 local-root 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 local-root.

    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 local-root. X's MRT-Red next-hops to W are X's increasing aka Blue MRT next-hops to X's local-root.

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 4.6.5.

4.6.5. Complete Algorithm to Compute MRT Next-Hops

The complete algorithm to compute MRT Next-Hops for a particular router X is given in Figure 22. 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.localroot is y.localroot) and (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, spf_root, store_nhs)
   if direction is FORWARD
      y.higher = true
      if store_nhs
         y.blue_next_hops = y.next_hops
   if direction is REVERSE
      y.lower = true
      if store_nhs
         y.red_next_hops = y.next_hops

SPF_No_Traverse_Root(spf_root, block_root, direction, store_nhs)
   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, spf_root, store_nhs)
       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 is 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
     if (y.local_root != y) {
        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, 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_Root(x, x.localroot, FORWARD, TRUE)
   SPF_No_Traverse_Root(x, x.localroot, REVERSE, TRUE)

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

   // Handle nodes in the same block that aren't the local-root
   foreach node y
     if (y.IN_MRT_ISLAND and (y is not x) and
         (y.localroot is x.localroot) and
         ((y is x.localroot) or (x is y.localroot) or
          (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 root
         root.blue_next_hops = x.localroot.blue_next_hops
         root.red_next_hops = x.localroot.red_next_hops
         root.order_proxy = x.localroot
      foreach node y
         if (y is not root) and (y is not x) and y.IN_MRT_ISLAND
           SetEdge(y)

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

Figure 22

4.7. 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 of the MRTs 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 next-hops as the alternate next-hop(s) for that primary next hop or to use the MRT-Red next-hops. The algorithm is given in Figure 23 and discussed afterwards.

```
   Select_Alternates_Internal(S, D, F, primary_intf,
                              D_lower, D_higher, D_topo_order)

       //When D==F, we can do only link protection
       if ((D is F) or (D.order_proxy is F))
```

```
            if an MRT doesn't use primary_intf
                indicate alternate is not node-protecting
                return that MRT color
            else // parallel links are cut-edge
                return AVOID_LINK_ON_BLUE

        if (D_lower and D_higher and F.lower and F.higher)
            if F.topo_order < D_topo_order
                return USE_RED
            else
                return USE_BLUE

        if (D_lower and D_higher)
            if F.higher
                return USE_RED
            else
                return USE_BLUE

        if (F.lower and F.higher)
            if D_lower
                return USE_RED
            else if D_higher
                return USE_BLUE
            else
                if primary_intf.OUTGOING and primary_intf.INCOMING
                    return AVOID_LINK_ON_BLUE
                if primary_intf.OUTGOING is true
                    return USE_BLUE
                if primary_intf.INCOMING is true
                    return USE_RED

        if D_higher
            if F.higher
                if F.topo_order < D_topo_order
                    return USE_RED
                else
                    return USE_BLUE
            else if F.lower
                return USE_BLUE
            else
                // F and S are neighbors so either F << S or F >> S
        else if D_lower
            if F.higher
```

```
            return USE_RED
        else if F.lower
            if F.topo_order < D_topo_order
                return USE_RED
            else
```



```
                return USE_BLUE
            else
                // F and S are neighbors so either F << S or F >> S
     else // D and S not ordered
        if F.lower
            return USE_RED
        else if F.higher
            return USE_BLUE
        else
            // F and S are neighbors so either F << S or F >> S


Select_Alternates(S, D, F, primary_intf)
   if D.order_proxy is not D
      D_lower = D.order_proxy.lower
      D_higher = D.order_proxy.higher
      D_topo_order = D.order_proxy.topo_order
   else
      D_lower = D.lower
      D_higher = D.higher
      D_topo_order = D.topo_order
   return Select_Alternates_Internal(S, D, F, primary_intf,
                                     D_lower, D_higher, D_topo_order)
```

Figure 23

If either D>>S>>F or D<<S<<F holds true, the situation is simple: in the first case we should choose the increasing Blue next-hop, in the second case, the decreasing Red next-hop is the right choice.

However, when both D and F are greater than S the situation is not so simple, there can be three possibilities: (i) F>>D (ii) F<<D or (iii) F and D are not ordered. In the first case, we should choose the path towards D along the Blue tree. In contrast, in case (ii) the Red path towards the root and then to D would be the solution. Finally, in case (iii) both paths would be acceptable. However,

observe that if e.g. F.topo_order>D.topo_order, either case (i) or case (iii) holds true, which means that selecting the Blue next-hop is safe. Similarly, if F.topo_order<D.topo_order, we should select the Red next-hop. The situation is almost the same if both F and D are less than S.

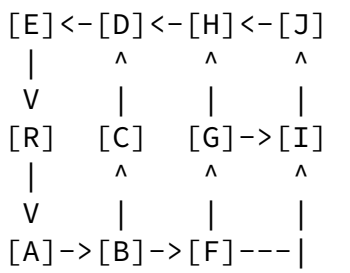
Recall that we have added each link to the GADAG in some direction, so it is impossible that S and F are not ordered. But it is possible that S and D are not ordered, so we need to deal with this case as well. If F<<S, we can use the Red next-hop, because that path is first increasing until a node definitely greater than D is reached, than decreasing; this path must avoid using F. Similarly, if F>>S, we should use the Blue next-hop.

Additionally, the cases where either F or D is ordered both higher and lower must be considered; this can happen when one is a block-root or its order_proxy is. If D is both higher and lower than S, then the MRT to use is the one that avoids F so if F is higher, then the MRT-Red should be used and if F is lower, then the MRT-Blue should be used; F and S must be ordered because they are neighbors. If F is both higher and lower, then if D is lower, using the MRT-Red to decrease reaches D and if D is higher, using the Blue MRT to increase reaches D; if D is unordered compared to S, then the situation is a bit more complicated.

In the case where F<<S<<F and D and S are unordered, the direction of the link in the GADAG between S and F should be examined. If the link is directed S -> F, then use the MRT-Blue (decrease to avoid that link and then increase). If the link is directed S <- F, then use the MRT-Red (increase to avoid that link and then decrease). If the link is S <--> F, then the link must be a cut-link and there is no node-protecting alternate. If there are multiple links between S and F, then they can protect against each other; of course, in this situation, they are probably already ECMP.

Finally, there is the case where D is also F. In this case, only link protection is possible. The MRT that doesn't use the indicated primary next-hop is used. If both MRTs use the primary next-hop, then the primary next-hop must be a cut-edge so either MRT could be used but the set of MRT next-hops must be pruned to avoid that primary next-hop. To indicate this case, Select_Alternates returns AVOID_LINK_ON_BLUE.

As an example, consider the ADAG depicted in Figure 24 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 not smaller than 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.



(a)
a 2-connected graph

Figure 24

4.8. Finding FRR Next-Hops for Proxy-Nodes

As discussed in Section 10.2 of [I-D.ietf-rtgwg-mrt-frr-architecture], it is necessary to find MRT-Blue and MRT-Red next-hops and MRT-FRR alternates for a named proxy-nodes. An example case is for a router that is not part of that local MRT Island, when there is only partial MRT support in the domain.

A first incorrect and naive approach to handling proxy-nodes, which cannot be transited, is to simply add these proxy-nodes to the graph of the network and connect it to the routers through which the new proxy-node can be reached. Unfortunately, this can introduce some new ordering between the border routers connected to the new node which could result in routing MRT paths through the proxy-node. Thus, this naive approach would need to recompute GADAGs and redo SPTs for each proxy-node.

Instead of adding the proxy-node to the original network graph, each individual proxy-node can be individually added to the GADAG. The proxy-node is connected to at most two nodes in the GADAG. Section 10.2 of [I-D.ietf-rtgwg-mrt-frr-architecture] defines how the proxy-node attachments MUST be determined. 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 attachment node; if there are different interfaces, then some can be assigned to MRT-Red and others to MRT_Blue.

Now, consider the proxy-node that is attached to exactly two nodes in the GADAG. Let the order_proxies of these nodes be A and B. Let the current node, where next-hop is just being calculated, be S. If one of these two nodes A and B is the local root of S, let A=S.local_root and the other one be B. Otherwise, let A.topo_order < B.topo_order.

A valid GADAG was constructed. Instead doing an increasing-SPF and a decreasing-SPF to find ordering for the proxy-nodes, the following simple rules, providing the same result, can be used independently for each different proxy-node. For the following rules, let X=A.local_root, and if A is the local root, let that be strictly lower than any other node. Always take the first rule that matches.

Rule	Condition	Blue NH	Red NH	Notes
1	S=X	Blue to A	Red to B	
2	S<<A	Blue to A	Red to R	
3	S>>B	Blue to R	Red to B	
4	A<<S<<B	Red to A	Blue to B	
5	A<<S	Red to A	Blue to R	S not ordered w/ B
6	S<<B	Red to R	Blue to B	S not ordered w/ A

7	Otherwise	Red to R	Blue to R	S not ordered w/ A+B

These rules are realized in the following pseudocode where P is the proxy-node, X and Y are the nodes that P is attached to, and S is the computing router:

```
    Select_Proxy_Node_NHs(P, S, X, Y)
        if (X.order_proxy.topo_order < Y.order_proxy.topo_order)
            //This fits even if X.order_proxy=S.local_root
            A=X.order_proxy
            B=Y.order_proxy
        else
            A=Y.order_proxy
            B=X.order_proxy

        if (S==A.local_root)
            P.blue_next_hops = A.blue_next_hops
            P.red_next_hops  = B.red_next_hops
            return
        if (A.higher)
            P.blue_next_hops = A.blue_next_hops
            P.red_next_hops  = R.red_next_hops
            return
        if (B.lower)
            P.blue_next_hops = R.blue_next_hops
            P.red_next_hops  = B.red_next_hops
            return
        if (A.lower && B.higher)
            P.blue_next_hops = A.red_next_hops
            P.red_next_hops  = B.blue_next_hops
            return
        if (A.lower)
```



```
            P.blue_next_hops = R.red_next_hops
            P.red_next_hops  = B.blue_next_hops
            return
        if (B.higher)
            P.blue_next_hops = A.red_next_hops
            P.red_next_hops  = R.blue_next_hops
            return
        P.blue_next_hops = R.red_next_hops
```

```
        P.red_next_hops  = R.blue_next_hops
        return
```

After finding the the red and the blue next-hops, it is necessary to know which one of these to use in the case of failure. This can be done by Select_Alternates_Inner(). In order to use Select_Alternates_Internal(), we need to know if P is greater, less or unordered with S, and P.topo_order. P.lower = B.lower, P.higher = A.higher, and any value is OK for P.topo_order, until A.topo_order<=P.topo_order<=B.topo_order and P.topo_order is not equal to the topo_order of the failed node. So for simplicity let P.topo_order=A.topo_order when the next-hop is not A, and P.topo_order=B.topo_order otherwise. This gives the following pseudo-code:

```
  Select_Alternates_Proxy_Node(S, P, F, primary_intf)
     if (F is not P.neighbor_A)
        return Select_Alternates_Internal(S, P, F, primary_intf,
                                          P.neighbor_B.lower,
                                          P.neighbor_A.higher,
                                          P.neighbor_A.topo_order)
     else
        return Select_Alternates_Internal(S, P, F, primary_intf,
                                          P.neighbor_B.lower,
                                          P.neighbor_A.higher,
                                          P.neighbor_B.topo_order)
```

Figure 25

5. MRT Lowpoint Algorithm: Complete Specification

This specification defines the MRT Lowpoint Algorithm, which include

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 4.6 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 >> S, 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.

In a future version, this section will include pseudo-code describing the full code path through the pseudo-code given earlier in the draft.

6. Algorithm Alternatives and Evaluation

This specification defines the MRT Lowpoint Algorithm, which is one option among several possible MRT algorithms. Other alternatives are described in the appendices.

In addition, it is possible to calculate Destination-Rooted GADAG, where for each destination, a GADAG rooted at that destination is computed. Then a router can compute the blue MRT and red MRT next-hops to that destination. Building GADAGs per destination is computationally more expensive, but may give somewhat shorter alternate paths. It may be useful for live-live multicast along MRTs.

6.1. Algorithm Evaluation

This section compares MRT and remote LFA for IP Fast Reroute in 16 service provider network topologies, focusing on coverage and alternate path length. Figure 26 shows the coverage provided by MRT and RLFA for protection against different failure modes in these topologies. The coverage values are calculated as the percentage of source-destination pairs protected by the given IPFRR method relative to source-destination pairs protected by optimal routing, against the same failure modes. For example, the second column is the percentage of source-destination pairs protected by MRT against next-hop node failure and against next-hop link failure relative to source-

destination pairs protected by optimal routing against the same failure modes. The particular variants of MRT and RLFA used for this analysis are described at the end of this section.

When the requirement is to provide IPFRR protection against a single link or node failure, MRT is able to provide protection for any source-destination pair for which some path still exists after the failure. For the topologies analyzed here, RLFA provides protection against a single link or node failure for 41% to 98% of the source-destination pairs, with an average of 73% coverage.

Topology	link and node failure coverage		link-only failure coverage		
	MRT	RLFA	MRT	RLFA no possible loops	RLFA possible loops
T101	100.0%	89.0%	100.0%	97.1%	99.4%
T102	100.0%	41.0%	100.0%	96.5%	100.0%
T103	100.0%	81.7%	100.0%	94.9%	99.6%
T104	100.0%	65.4%	100.0%	86.2%	100.0%
T105	100.0%	69.0%	100.0%	85.7%	93.8%
T106	100.0%	80.3%	100.0%	91.2%	100.0%
T107	100.0%	79.6%	100.0%	82.1%	93.7%
T108	100.0%	60.4%	100.0%	54.9%	66.9%
T109	100.0%	50.7%	100.0%	52.9%	67.0%
T110	100.0%	80.5%	100.0%	75.4%	100.0%
T111	100.0%	85.1%	100.0%	89.5%	99.9%
T112	100.0%	89.1%	100.0%	76.9%	100.0%
T113	100.0%	66.7%	100.0%	93.7%	100.0%
T114	100.0%	73.6%	100.0%	96.0%	100.0%
T115	100.0%	97.7%	100.0%	96.2%	100.0%
T116	100.0%	65.0%	100.0%	95.7%	99.9%
Average	100.0%	73.4%	100.0%	85.3%	95.0%
Median	100.0%	76.6%	100.0%	90.4%	100.0%

Figure 26

When the requirement is to provide protection against a single link failure, MRT is able to provide 100% coverage. The coverage provided by RLFA against a single link failure depends on whether or not one restricts RLFA repairs to those that are guaranteed not to cause loops in the event of a node failure occurs. When RLFAs are chosen

to exclude the possiblity of such loops, coverage for these topologies ranges from 52% to 97%, with an average of 85%. If one

allows for the possibility of loops being created by the use of an RLFA, then the coverage increases to range from 67% to 100%, with an average of 95%.

Note that for most of the topologies, the calculated RLFA coverage increases when reducing the protection requirements from link and node failure coverage to link-only failure coverage. However, for several of the topologies, the calculated RLFA coverage decreases when going from link and node failure coverage to link-only failure coverage. While the absolute number of source-destination pairs protected by RLFA increases for all of the topologies when the protection requirements are reduced, for some topologies the absolute number of source-destination pairs that protectable by optimal routing increases even more, resulting in a decrease in the relative RLFA coverage.

Topology	average alternate path length (relative to optimal)		
	MRT	RLFA	Next-Next-Hop
T101	1.28	1.01	1.04
T102	1.22	1.01	1.02
T103	1.13	1.01	1.02
T104	1.01	1.00	1.00
T105	2.97	1.42	1.16
T106	1.16	1.06	1.07
T107	86.87	99.51	1.04
T108	1.07	1.01	1.03
T109	1.09	1.02	1.05
T110	1.06	1.03	1.25
T111	1.25	1.02	1.10
T112	1.11	1.05	1.32
T113	1.03	1.00	1.02
T114	1.77	1.00	1.06
T115	1.01	1.00	1.04
T116	1.31	1.01	1.04
Median	1.14	1.01	1.04

+----------+---------------+---------------+---------------+

Figure 27

The first three columns of Figure 27 compare the lengths of the alternate paths used by MRT and RLFA across the 16 topologies (measured as the sum of IGP costs). The alternate path lengths for the FRR methods are computed relative to the optimal alternate path length, which is computed by removing the failed node and running an

SPF to find the shortest path from the PLR to the destination. The alternate path lengths are averaged over all source-destination pairs for which RLFA provides a node-protecting alternate or a link-protecting alternate that cannot loop in the event of a node failure. The fourth column of Figure 27 presents results for Next-Next-Hop alternate paths. The calculated Next-Next-Hop alternate lengths would apply to the Not-Via IPFRR method as well as RSVP-TE based FRR using next-next-hop bypass tunnels.

Before drawing any general conclusions from this data, it is useful understand the origin of the large values of average relative alternate path length calculated for topology T107, with a value of 87 for MRT, and 99 for RLFA. The network topology represented by T107 uses values of 10, 100, and 1000 as IGP costs, so small deviations from the optimal alternate path can result in large differences in relative path length. The fact that the Next-Next-Hop average relative alternate path length for T107 is 1.04 (much closer to optimal than either MRT or RLFA is reasonable. The next-next-hop alternate path length is computed by removing the failed node and finding the shortest path from the source to the next-next-hop node, and adding that to the shortest path from the next-next-hop node to the destination. Both of the paths that make up the Next-Next-Hop alternate path follow shortest paths, so the resulting alternate path from source to destination will only use a link with a cost of 1000 if absolutely necessary. Instead, the other two methods allow for at least one hop in the alterate path to be chosen independent of the cost of the link, often resulting in the use of a link with cost 1000.

For most of the topologies analyzed, the average RLFA alternate paths are within a few percent of optimal with a median value of 1.01. Most of the average MRT alternate path lengths are within 30% of

optimal with a median value of 1.14. In general, it appears that the 100% coverage provided by MRT comes at the expense of a modest increase in alternate path length. This may be a desirable trade-off if one considers that the alternate path length for a source-destination pair without an RLFA alternate is effectively infinite. The results for Next-Next-Hop alternate path lengths tend to fall in between MRT and RLFA with a median of 1.04. However, for some topologies the Next-Next-Hop results are higher than the MRT results.

The analysis presented here uses the MRT Lowpoint Algorithm defined in this specification with a common GADAG root. The particular choice of a common GADAG root is expected to affect the quality of the MRT alternate paths, with a more central common GADAG root resulting in shorter MRT alternate path lengths. For this analysis, a single GADAG root was chosen for each topology by calculating the sum of costs of all shortest paths to and from a given node. The

node with the lowest sum was chosen as the common GADAG root. In actual deployments, the common GADAG root would be chosen based on the GADAG Root Selection Priority advertised by each router, the values of which would be determined off-line.

Both MRT and remote LFA have the option of using a local LFA in the alternate selection process. The details of the alternate selection processes used in this analysis are provided below.

For the MRT analysis, for each source, destination, and primary next-hop, each alternate was chosen with the following priorty:

1. node-protecting downstream local LFA

2. node-protecting MRT alternate

3. link-protecting downstream local LFA

4. MRT alternate

For the RLFA analysis, each alternate was chosen with the following priority:

1. node-protecting local LFA

2. link-protecting remote LFA providing node-protection

3. link-protecting downstream local LFA

4. link-protecting downstream remote LFA

5. link-protecting local LFA

6. link-protecting remote LFA

7. Algorithm Work to Be Done

Broadcast Interfaces: The algorithm assumes that broadcast interfaces are already represented as pseudo-nodes in the network graph. Given maximal redundancy, one of the MRT will try to avoid both the pseudo-node and the next hop. The exact rules need to be fully specified.

8. IANA Considerations

This doument includes no request to IANA.

9. Security Considerations

This architecture is not currently believed to introduce new security concerns.

10. References

10.1. Normative References

[I-D.ietf-rtgwg-mrt-frr-architecture]
Atlas, A., Kebler, R., Envedi, G., Csaszar, A., Tantsura, J., Konstantynowicz, M., and R. White, "An Architecture for IP/LDP Fast-Reroute Using Maximally Redundant Trees", draft-ietf-rtgwg-mrt-frr-architecture-03 (work in

progress), July 2013.

10.2. 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, <http://www.omikk.bme.hu/collections/phd/Villamosmernoki_es_Informatikai_Kar/2011/Enyedi_Gabor/ertekezes.pdf>.

[I-D.atlas-ospf-mrt]
Atlas, A., Hegde, S., Chris, C., and J. Tantsura, "OSPF Extensions to Support Maximally Redundant Trees", draft-atlas-ospf-mrt-00 (work in progress), July 2013.

[I-D.ietf-rtgwg-ipfrr-notvia-addresses]
Bryant, S., Previdi, S., and M. Shand, "A Framework for IP and MPLS Fast Reroute Using Not-via Addresses", draft-ietf-rtgwg-ipfrr-notvia-addresses-11 (work in progress), May 2013.

[I-D.ietf-rtgwg-lfa-manageability]
Litkowski, S., Decraene, B., Filsfils, C., and K. Raza, "Operational management of Loop Free Alternates", draft-ietf-rtgwg-lfa-manageability-00 (work in progress), May 2013.

[I-D.ietf-rtgwg-remote-lfa]

Bryant, S., Filsfils, C., Previdi, S., Shand, M., and S. Ning, "Remote LFA FRR", draft-ietf-rtgwg-remote-lfa-02 (work in progress), May 2013.

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

[LFARevisited]
Retvari, G., Tapolcai, J., Enyedi, G., and A. Csaszar, "IP

Fast ReRoute: Loop Free Alternates Revisited", Proceedings of IEEE INFOCOM , 2011, <http://opti.tmit.bme.hu/~tapolcai/papers/retvari2011lfa_infocom.pdf>.

[LightweightNotVia]
Enyedi, G., Retvari, G., Szilagyi, P., and A. Csaszar, "IP Fast ReRoute: Lightweight Not-Via without Additional Addresses", Proceedings of IEEE INFOCOM , 2009, <http://mycite.omikk.bme.hu/doc/71691.pdf>.

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

[RFC3137] Retana, A., Nguyen, L., White, R., Zinin, A., and D. McPherson, "OSPF Stub Router Advertisement", RFC 3137, June 2001.

[RFC5286] Atlas, A. and A. Zinin, "Basic Specification for IP Fast Reroute: Loop-Free Alternates", RFC 5286, September 2008.

[RFC5714] Shand, M. and S. Bryant, "IP Fast Reroute Framework", RFC 5714, January 2010.

[RFC6571] Filsfils, C., Francois, P., Shand, M., Decraene, B., Uttaro, J., Leymann, N., and M. Horneffer, "Loop-Free Alternate (LFA) Applicability in Service Provider (SP) Networks", RFC 6571, June 2012.

Appendix A. Option 2: Computing GADAG using SPFs

The basic idea in this option 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 mininized IN_GADAG node back to the SPF root.

To do this, first all cut-vertices must be identified and local-roots 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 local-root 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 is true
           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 28: Modified SPF for GADAG computation

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 4.4, 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 algorithm, 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 algorithm.

```
   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 29: Algorithm to assign links of an ear direction

A goal of the algorithm 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, as in the algorithm previously described in Figure 14.

The algorithm ignores interfaces picked from the ordered tree 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 will be seen in Section 4.6. 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-edges are a special case because there is no point in doing an SPF on a block of 2 nodes. The algorithm identifies cut-edges simply as links where both ends of the link are cut-vertices. Cut-edges 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 is true)
               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-edges
              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 30: SPF-based GADAG algorithm

Appendix B. Option 3: Computing GADAG using a hybrid method

In this option, the idea is to combine the salient features of the above two options. 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 lentghs) 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 pre-determined. 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 31

```
find_spf_stack_ear(stack, x, y, xy_intf, block_root)
   if L(y) == D(y)
      // Special case for cut-edges
      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 31: Hybrid GADAG algorithm

Authors' Addresses

Gabor Sandor Enyedi (editor)
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 (editor)
Juniper Networks
10 Technology Park Drive
Westford, MA 01886
USA

Email: akatlas@juniper.net

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

Email: cbowers@juniper.net

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

Email: abishek@ece.arizona.edu