Algorithms for computing Maximally Redundant Trees for IP/LDP Fast-Reroute
draft-enyedi-rtgwg-mrt-frr-algorithm-00

Abstract

   A complete solution for IP and LDP Fast-Reroute using Maximally
   Redundant Trees is presented in
   [I-D.atlas-rtgwg-mrt-frr-architecture]. This document describes an
   algorithm that can be used to compute the necessary Maximally
   Redundant Trees and the associated next-hops.

Status of this Memo

Copyright Notice

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 Blue MRT and the Red MRT. A router which
experiences a local failure must also have pre-determined which
alternate to use. This document describes how to compute these three
things and the algorithm design decisions and rationale. The
algorithms are 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 Blue MRT and Red MRT in a consistent
fashion. This is the motivation for the detail in this document.

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 Blue MRT for forwarding
received traffic on the Blue MRT and primary next-hops for the Red
MRT for forwarding received traffic on the Red MRT.

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 provides 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 decision-
making is not captured in this document.

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

        (a)                    (b)                      (c)
  a 2-connected graph    Blue MRT towards R       Red MRT towards R
```
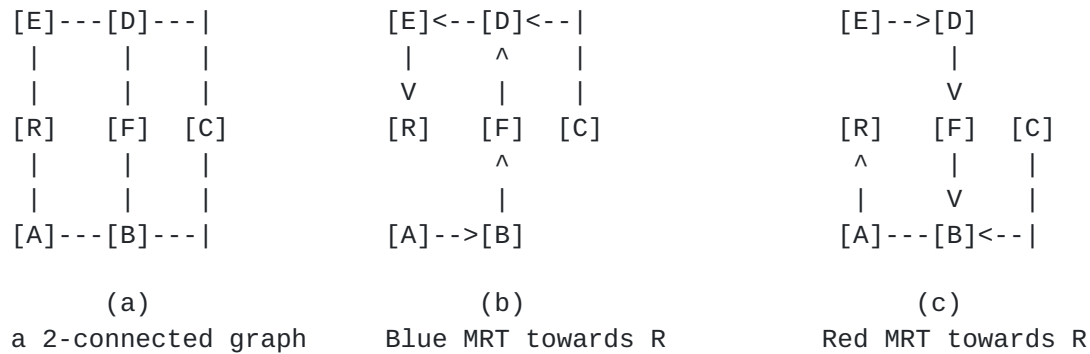
                               Figure 1

   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.

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

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

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

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

                               Figure 2

[2](#). **Terminology and Definitions**

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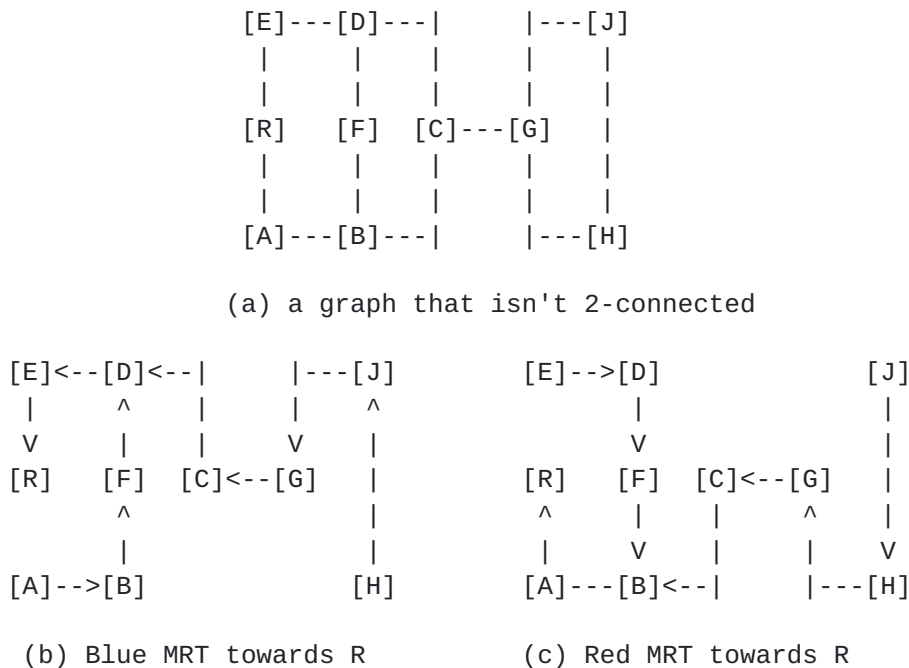   Any RT is an MRT but many MRTs are not RTs.

   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.

   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.

   DAG: Directed Acyclic Graph - a graph where all links are directed
      and there are no cycles in it.

   ADAG: Almost Directed Acyclic Graph - a graph that, if all links
      incoming to the root were removed, would be a DAG.

   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.

   GADAG: Generalized ADAG - a graph that is the combination of the
      ADAGs of all blocks.

   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.

[3](#). Algorithm Key Concepts

   There are five key concepts that are critical for understanding the
   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 IP
loopback addresses. In a partial order, there may be some nodes for
which it can't be determined whether X << Y or X >> Y. A partial
order can be captured in a directed graph, as shown in Figure 3. In
a graphical representation, a link directed from X to Y indicates
that X is a neighbor of Y in the network graph and X << Y.

```
    [A]<---[R]    [E]        R << A << B << C << D << E
     |            ^          R << A << B << F << G << H << D << E
     |            |
     V            |          Unspecified Relationships:
    [B]--->[C]--->[D]              C and F
     |            ^                C and G
     |            |                C and H
     V            |
    [F]--->[G]--->[H]
```
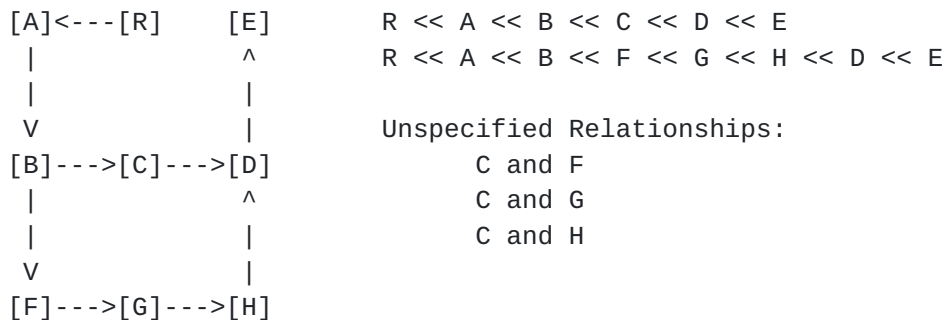
              Figure 3: Directed Graph showing a Partial Order

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

```
   [A]<---[R]<---[E]        R << A << B << C << R
    |      ^      ^         R << A << B << C << D << E << R
    |      |      |         R << A << B << F << G << H << D << E << R
    V      |      |
   [B]--->[C]--->[D]        Unspecified Relationships:
    |             ^                C and F
    |             |                C and G
    V             |                C and H
   [F]--->[G]--->[H]
```
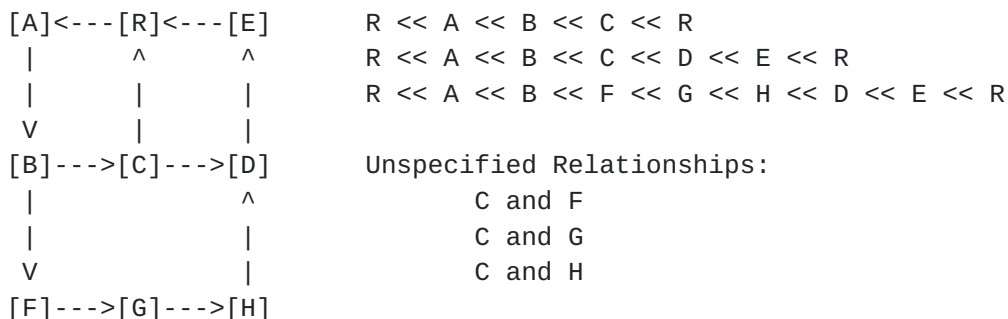
        Figure 4: ADAG showing a Partial Order with R lowest and highest

Most importantly, if a node Y >> X, then Y can only appear on the

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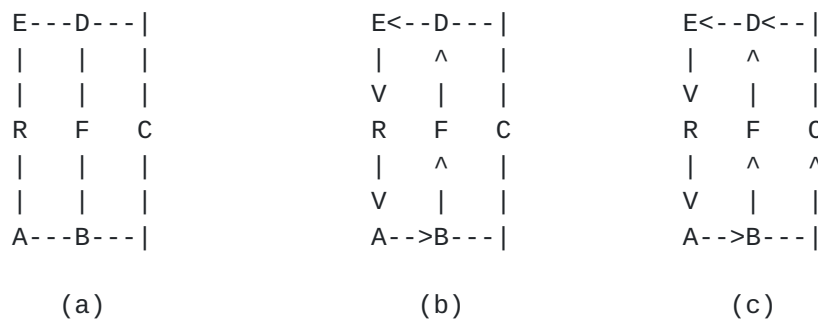path from X to the root and never on the inceasing path.

Additionally, 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 algorithms given in Section 4.3 and
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

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

      (a)                (b)                (c)
```

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

                         Figure 5

   In this partial ADAG, node C is not yet included. However, we can
   find path B-C-D, where both endpoints are contained by this partial
   ADAG (we say those nodes are *ready* in the 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 greater 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 the shortest cycle containing root.
    Add the shortest 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 (X is root) or (Y<<X)
          add the ear towards X to the ADAG
       else // (a) Y is root or (b) Y>>X 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. There are two options being considered. The Low-Point
Inheritance option is described in Section 4.3. The SPF-based option
is described in Section 4.4.

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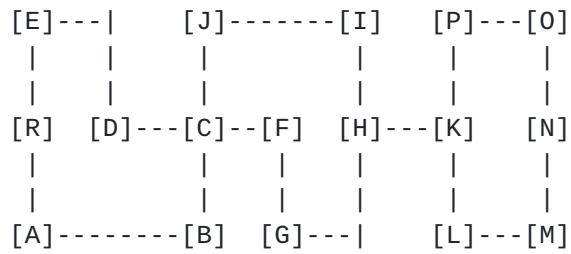
```
     [E]---|    [J]-------[I]   [P]---[O]
      |    |     |         |     |     |
      |    |     |         |     |     |
     [R]  [D]---[C]--[F]  [H]---[K]   [N]
      |         |    |    |     |     |
      |         |    |    |     |     |
     [A]--------[B]  [G]---|    [L]---[M]

        (a) a non-2-connected graph

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

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

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

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


                       Figure 8
```
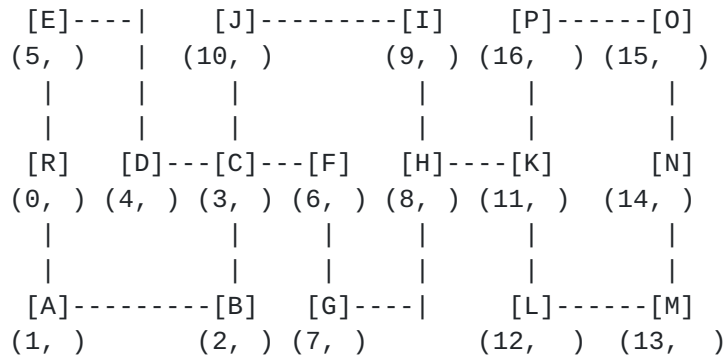
```
            global_variable: dfs_number

        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)
```
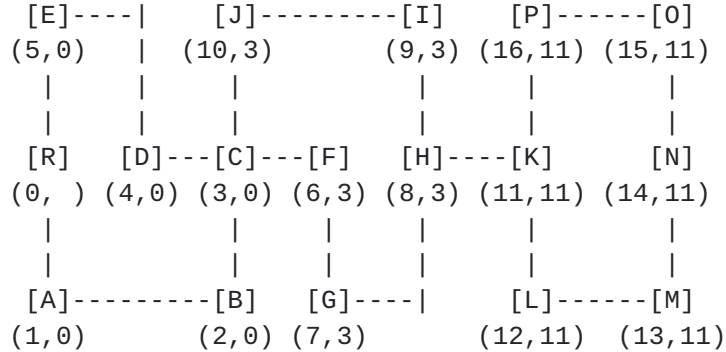
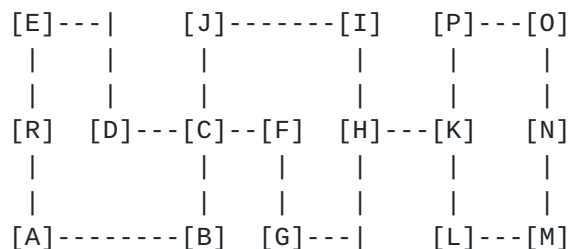   Figure 9: Computing Low-Point value

   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,
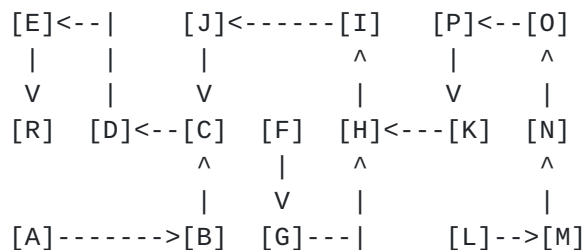   C's child D is in the same block as R while F is not.

   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
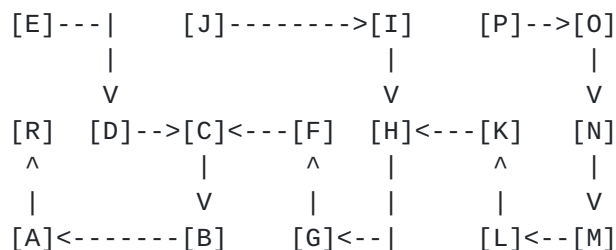   ears.

3.4. Blocks in a Graph

A key idea for the 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.

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


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



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


                    (b) Blue MRT



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


                    (c) Red MRT



                        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

   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 in the second option for
computing a GADAG using SPFs. The other method is used in the first
option 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 from s to a child c
               and then via lowpoint inheritance, e.g.
                 ( n = c
                   while n is not ready:
                       n = n.lowpoint_parent
                   e = n
                 )
               to a ready node e.
            if s is e
               s is a cut-vertex
               x.localroot = s
            else
               for each node x in the ear that is not s or e
                   x.localroot = s.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.

o 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.

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

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

4. Algorithm Sections

This algorithm computes one GADAG that is then used by a router to
determine its blue MRT and red MRT 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 13.

1. Select the root to use for the GADAG. [See Section 4.1.]

2. Initialize all interfaces to UNDIRECTED. [See Section 4.2.]

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

4. Construct the GADAG. [See Section 4.3 for Option 1 using
    Lowpoint Inheritance and Section 4.4 for Option 2 using SPFs.]

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

6. From the computing router x, compute the next-hops for the blue
    MRT and red MRT. [See Section 4.6.]

7. 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, it is necessary that all
routers 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 recommended ordering between two interfaces from the
same router x is given in Figure 13.

```
 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 ifindex.
   if a.ifindex < b.ifindex
      return A_LESS_THAN_B
   return B_LESS_THAN_A
```

Figure 13: Rules for ranking multiple interfaces. Order is from low
                           to high.

4.1. Root Selection

   The precise mechanism by which routers advertise a priority for the
   GADAG root is not described in this document. Nor is the algorithm
   for selecting routers based upon priority described in this document.

   A network may be partitioned or there may be islands of routers that
   support MRT fast-reroute. Therefore, the root selected for use in a
   GADAG must be consistent only across each connected island of MRT
   fast-reroute support. Before beginning computation, the network
   graph is reduced to contain only the set of routers that support a
   compatible MRT fast-reroute.

   The selection of a GADAG root is done among only those routers in the
   same MRT fast-reroute island as the computing router x.
   Additionally, only routers that are not marked as unusable or
   overloaded (e.g. ISIS overload or [RFC3137]) are eligible for
   selection as root.

4.2. 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, 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
   disussion is given of this restriction.

4.3. Option 1: Computing GADAG using lowpoint inheritance

   The basic idea of this is 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 will 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
to a cut-vertex to start and end at the same node.

Finally, the algorithm simultaneously computes each node's local-
root, as described in Figure 12. The local-root can be inherited
from the node x being processed to the nodes in the ear unless the
child of x is a cut-vertex in which case the rest of the nodes in the
ear are in a different block than x and have the child of x as their
local-root.

```
Construct_GADAG_via_Lowpoint(topology, root)
  root.IN_GADAG = true
  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_Ear(x, Stack, intf, type)
   ear_list = empty
   cur_node = intf.remote_node
   cur_intf = intf

   while cur_node.IN_GADAG is false
      cur_intf.UNDIRECTED = false
      cur_intf.OUTGOING = true
      cur_intf.remote_intf.UNDIRECTED = false
      cur_intf.remote_intf.INCOMING = true
      cur_node.IN_GADAG = true
      add_to_list_end(ear_list, cur_node)

      if type is CHILD
         cur_intf = cur_node.lowpoint_parent_intf
      else type must be NEIGHBOR
         cur_intf = cur_node.dfs_parent_intf
      cur_node = cur_intf.remote_node

    if (type is CHILD) and (cur_node is x)
       localroot = x
    else
       localroot = x.localroot
    while ear_list is not empty
       y = remove_end_item_from_list(ear_list)
       y.localroot = localroot
       push(Stack, y)
```

Figure 14: Low-point Inheritance GADAG algorithm

4.4. Option 2: Computing GADAG using SPFs

The basic idea in this option is to use slightly-modified SPF
computations to find ADAGs in each block. In each block, an SPF
computation is first done to find a cycle from the local root and
then SPF computations 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 11

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) and
                         (intf.remote_node 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(spf_root, block_root, ear_list, cut_vertex_list)
        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(cut_vertex_list, y)
          if y.local_node is a cut-vertex
            add_to_list_end(cut_vertex_list, y.local_node)
          y = y.local_node.spf_prev_intf
```

Figure 15: Modified SPF for GADAG computation

In Figure 15, while the path is determined, any non-end node in the
path that is a cut-vertex is added to the list of cut-vertices. This
ensures that there is a path from the GADAG root to that cut-vertex
before adding it to the list of nodes. All such cut-vertices will be
treated as the root of a block and the ADAG in that block will be
computed.

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 4.3, 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_a 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 16: 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 13.

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.

```
  Construct_GADAG_via_SPF(topology, root)
     Compute_Localroot(root, root)
     if root has multiple DFS-children
        mark root as a cut-vertex
     Initialize cut_vertex_list to empty
     Initialize ordered_intfs_tree to empty
     add_to_list_end(cut_vertex_list, root)
     while cut_vertex_list is not empty
        v = remove_start_item_from_list(cut_vertex_list)
        foreach interface intf of v
           if intf.remote_node is a cut-vertex
              // Special case for cut-edges
              intf.UNDIRECTED = false
              intf.remote_intf.UNDIRECTED = false
              intf.OUTGOING = true
              intf.INCOMING = true
              intf.remote_intf.OUTGOING = true
              intf.remote_intf.INCOMING = true
           else if intf.remote_node.localroot is v
              insert(ordered_intfs_tree, intf)
        v.IN_GADAG = true
        while ordered_intfs_trees is not empty
          cand_intf = remove_lowest(ordered_intfs_tree)
          if cand_intf.remote_node.IN_GADAG is false
             Mark all interfaces between cand_intf.remote_node
                and cand_intf.local_node as TEMP_UNUSABLE
             if cand_intf.local_node is not v
                Mark cand_intf.local_node as TEMP_UNUSABLE
             Initialize ear_list to empty
             ear_end = SPF_for_Ear(cand_intf.remote_node, v, ear_list,
                       cut_vertex_list)
             add_to_list_start(ear_list, cand_intf)
             Set_Ear_Direction(ear_list, cand_intf.remote, ear_end, v)
             Clear TEMP_UNUSABLE from all interfaces between
                cand_intf.remote_node and cand_intf.local_node
             Clear TEMP_UNUSABLE from cand_intf.local_node
```

                   Figure 17: SPF-based GADAG algorithm

4.5. Augmenting the GADAG by directing all links

   The GADAG, whether constructed via Low-Point Inheritance or with
   SPFs, 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 18

```
    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.INCOMING = true
                      i.remote_intf.OUTGOING = true
                      i.UNDIRECTED = false
                      i.remote_intf.UNDIRECTED = false
                   if i.INCOMING
                      if mark_or_clear is mark
                         i.TEMP_UNUSABLE = true
                         i.remote_intf.TEMP_UNUSABLE = true

    Run_Topological_Sort(topo, root)
       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

   Add_Undirected_Links(topo, root)
        Set_Block_Root_Incoming_Links(topo, root, MARK)
        Run_Topological_Sort(topo, root)
        Set_Block_Root_Incoming_Links(topo, root, CLEAR)
        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 18: Assigning direction to UNDIRECTED links

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 one pair of redundant trees with it, but a pair
   rooted at each node. This is ideal, since it is faster and it
   results packet forwarding easier to trace and/or debug. The method
   for doing that is based on two basic ideas. First, if two nodes X
   and Y are ordered with respect to each other in the partial order,
   then the same SPF and reverse-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 19,
if Y >> X, then there is an increasing path that goes from X to Y
without crossing R; this contains nodes in the interval [X,Y]. There
is also a decreasing path that decreases towards R and then decreases
from R to Y; this contains nodes in the interval [X,R-small] or
[R-great,Y]. The two paths cannot have common nodes other than X and
Y.

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

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

                   Figure 19: Y >> X
```
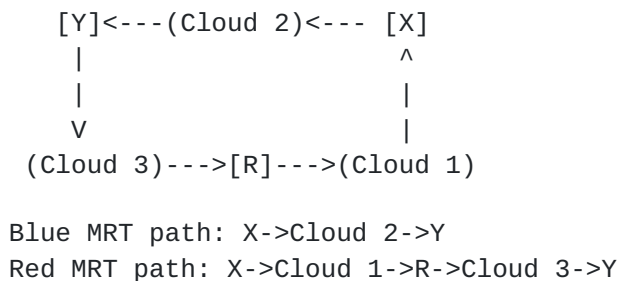
Similar logic applies if Y << X, as shown in Figure 20. In this
case, the increasing path from X increases to R and then increases
from R to Y to use nodes in the intervals [X,R-great] and [R-small,
Y]. The decreasing path from X reaches Y without crossing R and uses
nodes in the interval [Y,X].

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

          Blue MRT path: X->Cloud 3->R->Cloud 1->Y
          Red MRT path: X->Cloud 2->Y
```
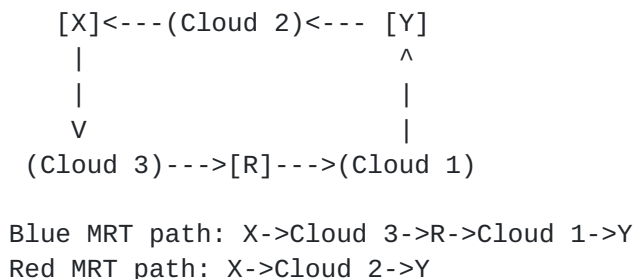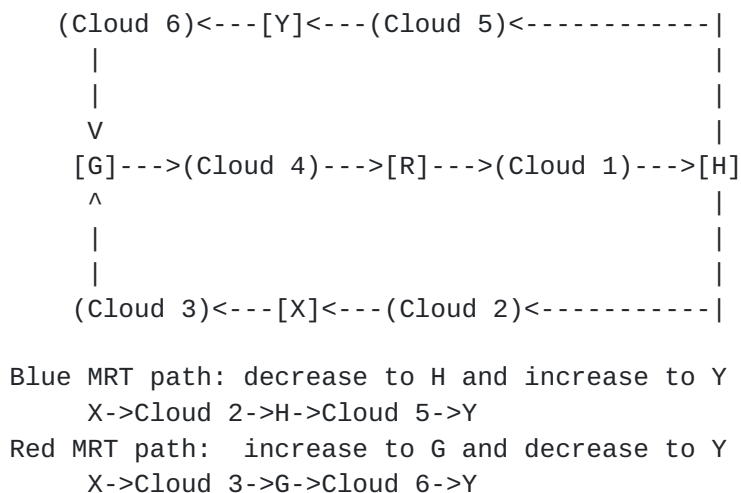
                       Figure 20: Y << X

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 21. It is
   necessary to decrease and then increase for the Blue MRT and increase
   and then decrease for the Red MRT; if one simply increased for one
   and decreased for the other, then both paths would go through the
   root R.

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

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

                     Figure 21: X and Y unordered

   This gives disjoint paths as long as G and H are not the same node.
   Since G >> Y and H << Y, if G and H could be the same node, that
   would have to be the root R. This is not possible because there is
   only one out-going interface from 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 towards R so that the associated interface on R is incoming
   and not outgoing. Therefore, there must be exactly one node M which
   is the smallest one after R, so the Blue MRT path will never reach R;
   it will turn at M and increase 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 SPF and a reverse-SPF rooted at X and not exploring any
   links from the ADAG root. (Other traversal algorithms could safely
   be used instead where one traversal takes the links in their given
   directions and the other reverses the links' directions.)

   An 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 Blue MRT to reach Y. A reverse-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 Red MRT to reach Z. Since the root R is both greater than and
   less than X, after this SPF and reverse-SPF, X's next-hops on the
   Blue MRT and on the Red MRT to reach R are known. For every node Y
   >> X, X's next-hops on the Red MRT to reach Y are set to those on the
   Red MRT to reach R. For every node Z << X, X's next-hops on the Blue
   MRT to reach Z are set to those on the Blue MRT to reach R.

   For those nodes, which were not reached, we have the next-hops as
   well. The increasing Blue MRT next-hop for a node, which is not
   ordered, is the next-hop along the decreasing Red MRT towards R and
   the decreasing Red MRT next-hop is the next-hop along the increasing
   Blue MRT 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 specific path taken
   but only 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 SPF
   rooted at R will reach all other nodes; the Blue MRT next-hops are
   those found with this SPF. Similarly, since the root R is >> all
   other nodes, running a reverse-SPF rooted at R will reach all other
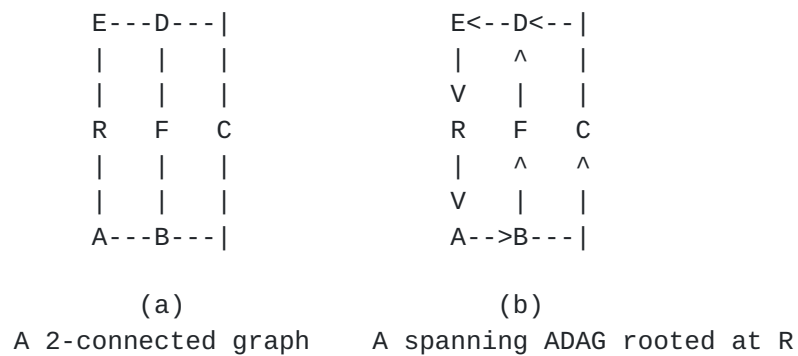   nodes; the Red MRT next-hops are those found with this reverse-SPF.

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


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


                           Figure 22
```

As an example consider the situation depicted in Figure 22. There
node C runs an SPF and a reverse-SPF The SPF reaches D, E and R and
the reverse SPF reaches B, A and R. So we immediately get that e.g.
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 increasing next-hop is the decreasing one towards R
(which is B) and the decreasing next-hop is the increasing one
towards R (which is D). Since B is ordered with F, it will find a
real increasing next-hop, so packet forwarded to B will get to F on
path C-B-F. Similarly, D will have a real decreasing next-hop, and
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 SPF
    and reverse-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 increasing SPF so Y >> X, then
          X's Red MRT next-hops to reach Y uses X's Red MRT next-hops
          to reach X's local-root and if Z <<, then X's Blue MRT next-
          hops to reach Z uses X's Blue MRT next-hops to reach X's
          local-root.

      C. If a node W in a common block with X was not reached in the
          SPF or reverse-SPF, then W is unordered with respect to X.
          X's Blue MRT next-hops to W are X's decreasing aka Red MRT
          next-hops to X's local-root. X's Red MRT 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 23. In addition to computing the Blue
   MRT next-hops and Red MRT 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 Blue MRT or the Red MRT can provide an
   acceptable alternate for a particular primary next-hop.

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

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

  SPF_No_Traverse_Root(spf_root, block_root, direction)
     Initialize spf_heap to empty
     Initialize nodes' spf_metric to infinity and next_hops to empty
     spf_root.spf_metric = 0
```

```
        insert(spf_heap, spf_root)
        while (spf_heap is not empty)
            min_node = remove_lowest(spf_heap)
            Store_Results(min_node, direction, spf_root)
            if 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))
                     if direction is FORWARD
                        path_metric = min_node.spf_metric + intf.metric
                     else
                        path_metric = min_node.spf_metric +
                                         intf.remote_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
        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)
      SPF_No_Traverse_Root(x, x.localroot, REVERSE)

      // red and blue next-hops are stored to x.localroot as different
      // paths are found via the SPF and reverse-SPF.
      // Similarly any 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 is not x) and (y.localroot is x.localroot)
             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)
             SetEdge(y)

     Copute_RT_NextHops(x, root)
```

<center>Figure 23</center>

4.7. Identify MRT alternates

At this point, a computing router S knows its Blue MRT next-hops and
Red MRT next-hops for each destination. 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) to determine whether to use the Blue MRT
next-hops as the alternate next-hop(s) for that primary next-hop or
to use the Red MRT next-hops. The algorithm is given in Figure 24
and discussed afterwards.

```
Select_Alternates(S, D, F)
    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

    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_BLUE
        else if F.upper
            return USE_RED
        else
            // F and S are neighbors so either F << S or F >> S
```

                             Figure 24

   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
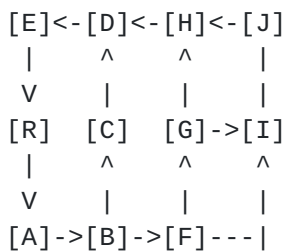   second case, the decreasing Red next-hop is the right choice.

   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 that is imposible 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.

As an example consider the ADAG depicted in Figure 25 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 first decrease to avoid using H, so the
Blue, first decreasing then increasing, path is selected.

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

                    (a)
              a 2-connected graph
```

                  Figure 25

5. **Algorithm Alternatives and Evaluation**

This description of the algorithm assumes a particular approach that
is believed to be a reasonable compromise between complexity and
computation. There are two options given for constructing the GADAG
as both are reasonable and promising.

   SPF-based GADAG Compute the common GADAG using Option 2 of SPF-based
      inheritance. This considers metrics when constructing the GADAG,
      which is important for path length and operational control. It
      has higher computational complexity than the Low-Point Inheritance
      GADAG.

   Low-Point Inheritance GADAG Compute the common GADAG using Option 1
      of Low-Point Inheritance. This ignores metrics when constructing
      the GADAG, but its computational complexity is O(links) which is
      attractive. It is possible that augmenting the GADAG by assigning
      directions to all links in the network graph and adding them to
      the GADAG will make the difference between this and the SPF-based
      GADAG minimal.

   In addition, it is possible to calculate Destination-Rooted GADAG,
   where for each destination, a GADAG rooted at that destination is
   computed. The GADAG can be computed using either Low-Point
   Inheritance or SPF-based. Then a router would need to 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.

5.1. Algorithm Evaluation

   When evaluating different algorithms and methods for IP Fast Reroute
   [RFC5714], there are three critical points to consider.

   o Coverage: For every Point of Local Repair (PLR) and local failure,
      is there an alternate to reach every destination? Those
      destinations include not only routers in the IGP area, but also
      prefixes outside the IGP area.

   o Alternate Length: What is the length of the alternate path offered
      compared to the optimal alternate route in the network? This is
      computed as the total length of the alternate path divided by the
      length of an optimal alternate path. The optimal alternate path
      is computed by removing the failed node and running an SPF to find
      the shortest path from the PLR to the destination.

   o Alternate Bandwidth: What percentage of the traffic sent to the
      failed point can be sent on the alternates? This is computed as
      the sum of the bandwidths along the alternate paths divided by the
      bandwidth of the primary paths that go through the failure point.

   Simulation and modeling to evelute the MRT algorithms is underway.
   The algorithms being compared are:

o SPF-based GADAG

o Low-Point Inheritance GADAG

o Destination-Rooted SPF-based GADAG

o Destination-Rooted Low-Point Inheritance GADAG

o Not-Via to Next-Next Hop[I-D.ietf-rtgwg-ipfrr-notvia-addresses]

o Loop-Free Alternates[RFC5286]

o Remote LFAs[I-D.shand-remote-lfa]

6. IANA Considerations

This doument includes no request to IANA.

7. Security Considerations

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

8. References

8.1. Normative References

[I-D.atlas-rtgwg-mrt-frr-architecture]
          Atlas, A., Konstantynowicz, M., Envedi, G., Csaszar, A.,
          White, R., and M. Shand, "An Architecture for IP/LDP Fast-
          Reroute Using Maximally Redundant Trees",
          draft-atlas-rtgwg-mrt-frr-architecture-00 (work in
          progress), July 2011.

8.2. 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://timon.tmit.bme.hu/theses/thesis_book.pdf>.

[I-D.ietf-rtgwg-ipfrr-notvia-addresses]
          Shand, M., Bryant, S., and S. Previdi, "IP Fast Reroute

              Using Not-via Addresses",
              draft-ietf-rtgwg-ipfrr-notvia-addresses-07 (work in
              progress), April 2011.

   [I-D.ietf-rtgwg-lfa-applicability]
              Filsfils, C., Francois, P., Shand, M., Decraene, B.,
              Uttaro, J., Leymann, N., and M. Horneffer, "LFA
              applicability in SP networks",
              draft-ietf-rtgwg-lfa-applicability-03 (work in progress),
              August 2011.

   [I-D.shand-remote-lfa]
              Bryant, S., Filsfils, C., Shand, M., and N. So, "Remote
              LFA FRR", draft-shand-remote-lfa-00 (work in progress),
              October 2011.

   [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.

Authors' Addresses

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

   Email: akatlas@juniper.net

   Gabor Sandor Enyedi
   Ericsson
   Konyves Kalman krt 11
   Budapest 1097
   Hungary

   Email: Gabor.Sandor.Enyedi@ericsson.com

   Andras Csaszar
   Ericsson
   Konyves Kalman krt 11
   Budapest 1097
   Hungary

   Email: Andras.Csaszar@ericsson.com