Energy-Saving Routing Algorithm using Steiner Tree

Hiroshi Matsuura
NTT Network Technology Laboratories
3-9-11, Midori-Cho, Musashino-Shi, Tokyo 180-8585, Japan
matsuura.hiroshi@lab.ntt.co.jp

***Abstract*— There is much demand for reducing energy consumption with regards to network communications. For this purpose, this paper proposes an energy-saving routing algorithm that uses a Steiner tree created using a branch-based multi-cast Steiner tree algorithm. The proposed algorithm creates a Steiner tree among edge nodes in a network and creates point to point paths between two different edge nodes by following the created Steiner tree. Since only the links and nodes on the Steiner tree are used, the numbers of used links and nodes are dramatically reduced compared with other routing algorithms. The proposed algorithm creates bypass routes between nodes on the Steiner tree to reduce traffic congestion between nodes. Many energy-saving routing algorithms have been proposed recently, but most are nondeterministic polynomial time (NP)-complete or NP-hard; thus, it is difficult to apply them to real-time routing operations. This paper compares the proposed routing algorithm with a polynomial time routing algorithm, which has already been proposed for energy saving, and the conventional open shortest path first (OSPF)-based routing algorithm to clarify the applicability of the proposed algorithm.**

I. Introduction

Saving energy with regards to network communication by taking into account the radical increase in network links and nodes is vital for reducing green house gases. Each network node or link has a sleep mode, which can drastically reduce energy consumption when it is not used for data transmission; therefore, it is important to increase the number of nodes and links that are in sleeping mode.

For this research, a routing server was used for energy saving routing. In the routing server, the proposed Steiner-tree-based energy saving routing algorithm was implemented to find the routes among edge nodes. Different from other centralized routing algorithms [1-5], the proposed algorithm runs in polynomial time, and it is even faster than the case in which Dijkstra's algorithm [12] is used for seeking shortest paths among all the edge nodes in a network. This fast routing of the proposed algorithm makes it possible to change the routes of energy-saving routing according to the network topology, network disorder or congestion, and appearance and disappearance of edge nodes.

In addition to being fast, the proposed algorithm does not use a traffic matrix among edge nodes, so it does not have to change the routes since traffic demand among edge nodes changes over time. The algorithm uses only the link costs of the network, so it does not require a special traffic engineering link state advertisement (TE LSA) [6], which is required for the algorithm proposed by Wang et al. [8].

The Steiner tree branch-based multi-cast (BBMC) algorithm [13] is used for the proposed energy saving routing because it produces the same Steiner tree created using the widely used minimum-cost path heuristic (MPH) algorithm [14], but it is much faster. After the creation of a Steiner tree among all the edge nodes in the network, the proposed algorithm creates point-to-point routes by using only the nodes and links on the created Steiner tree. Bypass routes are also proposed that replace the long ineffective routes if necessary after the creation of all the point-to-point routes.

Section II discusses related work. Section III discusses the software module architecture within the routing server explained in this paper. Section IV describes the proposed energy-saving routing algorithm, including its time complexities, compared with other current algorithms, namely shortest path tree (SPT)-based and energy-aware routing (EAR) [11] algorithms. Section V compares the numbers of sleeping nodes and links with each algorithm. The average hop-count and maximally used bandwidth (max bandwidth) in a link after route setting are also compared as well as the processing time of each algorithm. Section VI concludes this paper.

II. Related Work

Many energy-saving routing algorithms have been proposed [1-4] in which the total energy consumption used by network nodes and links is minimized on the condition that each link bandwidth threshold is not used more than its capacity. Vasic and Kostic [5] focused on future link hardware in which multiple transmission rates can be chosen so that, according to different traffic demands, its transmission rate can be changed. In this case, a link with a smaller transmission rate requires less energy, so not only the numbers of sleeping nodes and links but also each link's transmission rate are taken into account.

However, these centralized routing algorithms require a large amount of routing time because they have to compare almost all the possible routes among edge nodes. A heuristic algorithm for reducing the calculation time was proposed for each of these centralized routing algorithms, but they are not polynomial time algorithms; thus, a large routing delay is inevitable.

In addition, they try to optimize the energy consumption of a network on the condition that the traffic matrix among edge nodes is fixed. In this case, it is impossible to apply these routing algorithms to a real-time operation because the traffic matrix among edge nodes changes by time in a real network.

To overcome these problems, there have been studies on extending the open shortest path first (OSPF) [6] protocol for conserving energy. Energy saving on generalized multiprotocol label switching (GMPLS) [7] has been proposed [8] in which

978-3-901882-50-0

green energy sources are differentiated from dirty energy sources and network nodes powered by a green energy source has a lower cost, which is advertised by TE LSA. However, this study did not take into account how to increase the number of sleeping nodes and links.

Network connectivity [9] has been applied to the OSPF protocol for increasing the number of sleeping links [10]. The number of active links in a network can be reduced while maintaining some network robustness [10]. However, this method cannot take the topology of edge nodes into account, and the number of sleeping nodes cannot be increased.

Another OSPF-based distributed energy-saving routing algorithm is EAR, which uses an "exportation" mechanism [11]. The nodes that have a high node degree, which is the number of connected links to the node, become "exporters" and the neighboring nodes of an "exporter" become "importers" and import the SPT from the neighboring "exporter". Since an "importer" uses the routes to other nodes by following the SPT of its "exporter", there is a high possibility that the same links are repeatedly used, resulting in an increase in the number of sleeping links. This routing algorithm can be also applied to the centralized routing server solution; therefore, EAR is compared with the proposed algorithm in the following sections.

III. Routing Server Module Architecture

The proposed energy-saving routing algorithm can be implemented in each node by using TE LSA. However, it is more efficient if the algorithm is implemented in the centralized routing server because one Steiner tree among all the edge nodes is sufficient enough to create all the point-to-point routes among the edge nodes. In addition, the recognition of a new edge node or a node that is no longer functioning as an edge node is difficult for each node.

After the determination of the routes among edge nodes, these routes are set in the network by using the path computation element protocol (PCEP) [15] for a GMPLS or MPLS network, and by using the OpenFlow [16] protocol for an IP network.

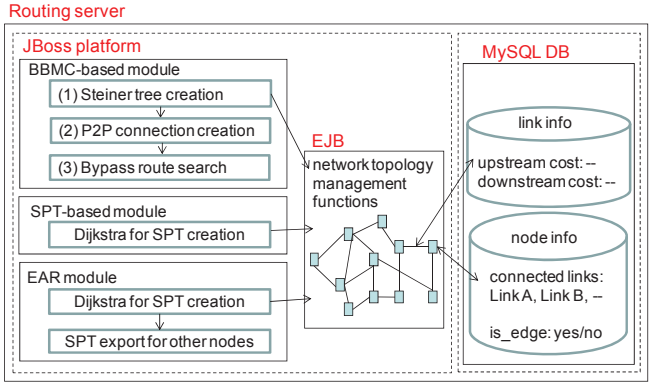

Figure 1. Routing server software modules

Figure 1 shows the software modules implemented in the routing server on a Linux machine. Three routing modules were implemented on a Java-based JBoss platform [17]. The BBMC-based module includes the proposed energy-saving routing algorithm, which consists of three processes. (1) The "Steiner tree creation" process accesses one of the network topology management functions and obtains link and node information to create a Steiner tree, which includes all the edge nodes, by using the BBMC algorithm. (2) The "P2P connection creation" process creates point-to-point routes among all the edge nodes from the created BBMC Steiner tree. This means only the nodes and links on the Steiner tree are used to determine these routes. (3) The "bypass route search" process creates bypass routes that substitute large hop-count routes and finalizes the routing algorithm. These processes are explained in detail in the next section.

We implemented two polynomial-time routing algorithms, which use OSPF-based routing, as the counterparts of the proposed routing algorithm. The "SPT-based module" includes Dijkstra's algorithm (a polynomial-time algorithm), which calculates an SPT from one edge node to the other edge nodes by using the network topology information. Dijkstra's algorithm has to be used as many times as the number of edge nodes because from all the edge nodes, the routes to other edge nodes should be calculated. The worst case time complexity of Dijkstra's algorithm to create an SPT is $O(l + n \log n)$, where l is the number of links and n is the number of nodes when Fibonacci heaps (F-heaps) are used [18]. Therefore, this SPT-based routing algorithm has $O(m (l + n \log n))$ worst case time complexity, where m is the number of edge nodes. The average case time complexity of this algorithm is the same as the worst case time complexity.

The EAR module runs the EAR algorithm. In the algorithm, "importers" use the SPT of a neighboring "exporter"; thus, the number of calls of the Dijkstra's algorithm is the same as the number of "exporters". Therefore, the worst and average case time complexities of EAR are $O(h (l + n \log n))$, where h is the number of "exporters".

Enterprise Java Beans (EJB) [19] was used to seamlessly access the MySQL database (DB) [20] by using Java functions without using database SQL commands. The MySQL DB stored data were used repeatedly by the three different routing modules and contained network topology information consisting of link and node info. Link info was composed of upstream and downstream link costs. Node info was composed of connecting links as Java pointers and also a java Boolean type attribute called "is_edge", which was "yes" when the node was an edge node; otherwise, it was "no". The information in the MySQL DB was used by the three routing algorithms: it was used by the BBMC-based module to create a Steiner tree and by the SPT-based and EAR modules to create an SPT from one edge node.

IV. Steiner-Tree-Based Energy-Saving Routing

In this section, the proposed Steiner-tree-based routing algorithm is described in detail. First, its pseudo code is explained and the "P2P connection creation" and "bypass route search" processes in the algorithm are discussed with examples. The worst and average case time complexities for the algorithm are also discussed.

A. Steiner-tree-based routing algorithm pseudo code

Figure 2 shows the pseudo code of the proposed Steiner-tree-based routing algorithm. The inputs to the algorithm are *s*, which is the root node of the Steiner tree, and *T*, which is the group of end nodes of the Steiner tree. "Radius" is also included in the inputs and is used for determining bypass routes in the algorithm. One of the edge nodes that has the most node degree becomes *s*, and the other edge nodes compose *T*. The reason *s* has the largest node degree is that a Steiner tree tends to have relatively more links from the root node. The output of the algorithm is *P2P_R*, which is the group of point-to-point routes among edge nodes.

```
Algorithm: Steiner_base_routing(s, T, radius)
Output: P2P_R

1   R = BBMC(s, T)
2   create_tree (R)
3   E = s U T, P2P_R = {};
4   FOR every e ∈E
5    P2P_R. addAll( create_p2p_route(e))
6   END FOR
7   binary_tree = list_routes_to_binary_tree (P2P_R)
8   base_bypass = null
9   WHILE(base_bypass = null)
10  route = get_longest_route(binary_tree)
11  shortest_route = dijkstra(route.start_node, route.end_node)
12  IF(is_independent (shortest_route) = true)
13     base_bypass = shortest_route    ENDIF
14     binary_tree.remove(route)
15  END WHILE
16  to_start = {}, from_start = {}, to_end = {}, from_end = {}
17  route = get_shortest_route(binary_tree)
18  WHILE(|route| ≦ radius)
19  IF(route.end_node.equals(base_bypass.start_node))
20     to_start.add(route) ENDIF
21  ELSE IF(route.start_node.equals(base_bypass.start_node))
22     from_start.add(route) ENDIF
23  ELSE IF(route.end_node.equals(base_bypass.end_node))
24     to_end.add(route) ENDIF
25  ELSE IF(route.start_node.equals(base_bypass.end_node))
26     from_end.add(route) ENDIF
27  binary_tree.remove(route)
28  route = get_shortest_route(binary_tree)
29  END WHILE
30  bypasses = {}
31  bypasses. addAll( create_bypass(to_start, base_bypass, from_end))
32  bypasses. addAll( create_bypass(to_end, r(base_bypass), from_start))
33  remove_replaced_routes( P2P_R)
34  P2P_R.addAll(bypasses),          return P2P_R
```

Figure 2. Steiner-tree-based routing algorithm pseudo code

The procedure from lines 1 to 2 corresponds to the "Steiner tree creation" process shown in Figure 1. At line 1 of the pseudo code, the algorithm requests the BBMC algorithm to return *R*, which is the group of branches composing a Steiner tree. At line 2, the Steiner tree is created by the "create_tree" function by using all the branches in *R*.

Figure 3 is an example of how a Steiner tree is constructed with the "create_tree" function. In this example, there are eight edge nodes except for *s*, and for each edge node from *e*1 to *e*8, there is a branch created using the BBMC algorithm. Therefore, there are eight branches from *b*1 to *b*8, each of which starts with an edge node and ends with another edge node via zero or more core nodes. The "create_tree" function creates relationships between a node and its connecting links on the Steiner tree. For example, there are three links connecting *s* with three different nodes *e*1, *e*2, and *e*3. The core node *c*1 also has three links connecting it with *e*1, *e*4, and *e*5.

In the implementation, these links had a relationship with the connected node by using Java objects, and the node Java object had pointers to the connected link Java objects. These Java objects were allocated on memory space in the JBoss platform and were not stored in the MySQL DB because they were used transiently only in this algorithm.

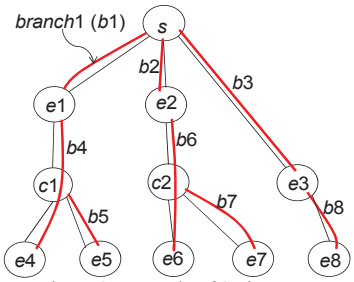

Figure 3. Example of Steiner tree

The procedure from lines 3 to 6 corresponds to the "P2P connection creation" process shown in Figure 1. At line 3, *E* is the group of all edge nodes, so *s* is added to *T*. *P2P_R* is set to an empty group for the point-to-point routes among the edge nodes.

The FOR routine from lines 4 to 6 is repeated as many times as the number of edge nodes. At line 5, newly created routes, which start from $e \in E$ and end with another edge node, are added to *P2P_R*. The "create_p2p_route (*e*)" function finds all the routes from *e* to other edge nodes by following the Steiner tree created by the "create_tree" function at line 2.

In Figure 3, for example, the "create_p2p_route(*s*)" function searches the routes from *s* by first following the links from *s,* and this search continues until all the edge nodes are reached. During the search from one edge node, the number of nodes gone through the search is less than the number of nodes in the network, and the search is repeated as many times as the number of edge nodes in the FOR routine from lines 4 to 6. Therefore, the worst case time complexity for the whole search is $O(mn)$, where *m* is the number of edge nodes and *n* is the number of nodes in the network.

At the end of line 6, all the routes among the edge nodes are set to *P2P_R*; thus, it is fine to finish the algorithm here. In reality, if there is no positive integer set to the "radius" as an input of the algorithm, the algorithm returns *P2P_R* as the output after line 6. However, there is a possibility that inefficient routes are included in *P2P_R*. Therefore, bypass routes can be optionally used to replace these inefficient routes if the "radius" is set to a positive integer. The procedure from lines 7 to 33 corresponds to the "bypass route search" process in Figure 1.

At line 7, all the routes in *P2P_R* are listed to the binary search tree "*binary_tree*" by the "list_routes_to_binary_tree" function based on their hop-counts. At line 8, the "base-bypass" is set to null before it is determined within the following WHILE routine.

The WHILE routine from lines 8 to 15 continues until the "*base_bypass*" is substituted with the "*shortest_route*" at line

13. At line 10, one of the longest routes (having the longest hop-count), is selected from the "*binary_tree*" as the "*route*". At line 11, between the starting node and ending node of the "*route*", the shortest route "*shortst_route*" is sought using Dijkstra's algorithm. At line 12, the "is_dependent(*shortest_route*)" function determines if there is a link on the "*shortest_route*", which is already used by the Steiner tree. If there is one, the algorithm again returns to line 9 after removing the "*route*" from the "*binary_tree*" at line 14. Otherwise, the "*base_bypass*" is substituted with the "*shortest_route*" at line 13, and the algorithm leaves the WHILE routine and goes to line 16.

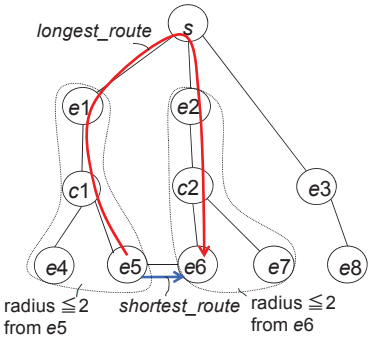

Figure 4. Bypass creation process example

An example of this WHILE routine process is shown in Figure 4. There are routes with a hop-count of 6 among the nine edge nodes on the Steiner tree shown in the figure. Suppose that the "*route*" from *e*4 to *e*6, which takes the route of *e*4-*c*1-*e*1-*s*-*e*2-*c*2-*e*6, is chosen at line 10 in the first cycle in the WHILE routine among these longest hop-count routes. Suppose also that at line 11, the "*shortest_route*" from *e*4 to *e*6 takes the route of *e*4-*c*1-*e*5-*e*6. In this case, the link between *e*4 and *c*1 is already used by the Steiner tree, so the "*route*" is removed from the "*binary_tree*" at line 14, and the algorithm again goes to line 9.

In the second cycle of the WHILE routine, "*base_bypass* is still set to null, so the algorithm goes to line 10 again. At line 10, suppose that the "*route*" from *e*5 to *e*6, which takes the route of *e*5-*c*1-*e*1-*s*-*e*2-*c*2-*e*6, is chosen. There is a direct link between *e*5 and *e*6, as shown in Figure 4, so the "*shortest_route*" route of *e*5-*e*6 is chosen at line 11. As shown in Figure 3, the Steiner tree does not use the link between *e*5 and *e*6, so the "*shortest_route*" does not have links that are used by the Steiner tree. Thus, at line 13, "*base_bypass*" is substituted with the route of *e*5-*e*6. After removing the "*route*" from the "*binary_tree*" at line 14, the algorithm goes to line 16.

At line 16, four route groups named "*to_start*", "*from_start*", "*to_end*", and "*from_end*" are initialized and used in the following WHILE routine from lines 18 to 29. At line 17, the "get_shortest_route(*binary_tree*)" function selects one of the shortest hop-count routes in the "*binary_tree*", and the "*route*" is substituted with the selected route.

As shown at line 18, the WHILE routine is repeated while the hop-count of the "*route*" is equal to or less than the "*radius*", which is a positive integer specified by the algorithm's input. If the "*route*" ends with the starting node of "*base_bypass*" at line 19, the "*route*" is added to "*to_start*". If the "*route*" starts with the starting node of "*base_bypass*" at line 21, the "*route*" is added to "*from_start*". If the "*route*" ends with the ending node of the "*base_bypass*" at line 23, the "*route*" is added to "*to_end*". If the "*route*" starts with the ending node of the "*base_bypass*" at line 25, the "*route*" is added to "*from_end*". The "*route*" is removed from the "*binary_tree*" at line 27, and is newly substituted with the route selected by the "get_shortest_route(*binary_tree*)" function at line 28. The algorithm then goes to the next cycle of the WHILE routine.

At the end of the WHILE routine, all the routes that have the same termination node as that of "*base_bypass*" and whose hop-count is equal to or less than the "*radius*" are added to one of the four route groups initialized at line 16. In Figure 4, when the "*radius*" is set to 2, "*to_start*" has 2 routes: *e*1-*c*1-*e*5 and *e*4-*c*1-*e*5, "*from_start*" has 2 routes: *e*5-*c*1-*e*1 and *e*5-*c*1-*e*4, "*to-end*" has 2 routes: *e*2-*c*2-*e*6 and *e*7-*c*2-*e*6, and "*from_end*" has 2 routes: *e*6-*c*2-*e*2 and *e*6-*c*2-*e*7.

At line 30, the bypass group "*bypasses*" is initialized. At line 31, the "create_bypass" function creates bypass routes by concatenating "*to_start*", "*base_bypass*", and "*from_end*", and these created bypass routes are added to "*bypasses*". In Figure 4, nine bypass routes are added to "*bypasses*" at line 31. These bypass routes are *e*1-*c*1-*e*5-*e*6, *e*1-*c*1-*e*5-*e*6-*c*2-*e*2, *e*1-*c*1-*e*5-*e*6-*c*2-*e*7, *e*4-*c*1-*e*5-*e*6, *e*4-*c*1-*e*5-*e*6-*c*2-*e*2, *e*4-*c*1-*e*5-*e*6-*c*2-*e*7, *e*5-*e*6, *e*5-*e*6-*c*2-*e*2, and *e*5-*e*6-*c*2-*e*7.

At line 32, the "create_bypass" function creates bypass routes for the opposite direction by concatenating "*to_end*", "r(*base_bypass*)", and "*from_start*". Here, "r(*base_bypass*)" is the reversed route of "*base_bypass*". In Figure 4, the "*base_bypass*" is *e*5-*e*6, so the "r(*base_bypass*)" is *e*6-*e*5. As a result, nine bypass routes are added to "*bypasses*". These bypass routes are *e*2-*c*2-*e*6-*e*5, *e*2-*c*2-*e*6-*e*5-*c*1-*e*1, *e*2-*c*2-*e*6-*e*5-*c*1-*e*4, *e*7-*c*2-*e*6-*e*5, *e*7-*c*2-*e*6-*e*5-*c*1-*e*1, *e*7-*c*2-*e*6-*e*5-*c*1-*e*4, *e*6-*e*5, *e*6-*e*5-*c*1-*e*1, and *e*6-*e*5-*c*1-*e*4.

At line 33, all the routes, which have the same starting and ending edge nodes with the bypass routes and thus will be replaced by the bypass routes, are removed from *P2P_R* by using the "remove_replaced_routes" function. At line 34, all the bypass routes within "*bypasses*" are added to *P2P_R*, and this *P2P_R* is returned as the output of the algorithm.

B. Steiner-tree-based routing algorithm's time complexities

In this subsection, the worst and average case time complexities of the proposed Steiner-tree-based energy-saving routing algorithm are discussed and compared with those of the SPT-based and EAR algorithms.

As Matsuura discussed [13], BBMC's worst case time complexity is $O(m(l + n \log n))$ and its average case time complexity is $O(\log m(l+ n \log n))$. As discussed in the previous subsection, the FOR routine from lines 4 to 6 of the algorithm takes $O(mn)$ time. Therefore, the worst case time complexity of the algorithm without bypass is $O(m(l+ n \log n))$, whereas its average case time complexity is $O(\log m(l + n \log n) + mn)$.

The dominant process of creating bypass routes from lines 7 to 34 to determine its time complexities is the WHILE routine

from lines 9 to 15. In this WHILE routine, Dijkstra's algorithm runs up to $m(m\text{-}1)$ times until the "*base_bypass*" is found. Therefore, the average and worst case time complexities for creating bypass routes are both $O(m^2(l + n \log n))$. However, the "*base_bypass*" is generally found in the early cycle of the WHILE routine; thus, real processing time is not as long as this time complexity, as explained by the processing time evaluation in the next section.

Table 1 compares the time complexities among three algorithms. The time complexities of EAR are smaller than the others when h, which is the number of "exporters", is smaller than m. The average case time complexity of the proposed Steiner-tree-based algorithm without using bypass routes is relatively smaller compared with the others. The time complexities of the Steiner-tree-based algorithm by using bypass routes are larger than the others.

Table 1. Time complexity comparison among algorithms

	average complexity	worst complexity
SPT-based	$O(m\ (l + n \log n))$	
EAR	$O(h\ (l + n \log n))$	
Steiner tree based without bypass	$O(\log m(l + n \log n) + mn)$	$O(m\ (l + n \log n))$
Steiner tree based with bypass	$O(m^2(l + n \log n))$	

V. Evaluations

The three routing algorithms, SPT based, EAR, and the proposed Steiner-tree based, were compared on three different evaluation networks. The first network was a sample network based on the MCI network topology [21], the second was the GEANT network [22], and the third network was randomly created using Waxman's model [23].

After creating point-to-point routes among all the edge nodes on these three networks, the numbers of sleeping links and nodes, average hop-counts of the routes, and max bandwidth in a link were compared among the algorithms. In addition, in the network created using Waxman's model, the processing times of the SPT-based algorithm were compared with those of the proposed Steiner-tree-based algorithm.

The common evaluation conditions are as follows. The routing server used in this evaluation was that shown in Figure 1. In addition, each link-cost of a directed link was determined as the inverse of the available bandwidth of the directed link. A node was put into sleeping mode when there was no traffic passing through the node. A link was put into sleep mode only if there was no traffic in both directions, upstream and downstream because a transceiver of upstream and downstream traffic generally does not have separate power control.

From the evaluations, the applicability of the proposed Steiner-tree-based algorithm to real-time network operations is now discussed.

A. Evaluation on sample network

The sample network used is shown in Figure 5, and it has 19 nodes and 31 links. First, seven nodes, r1, r2, r6, 7, r11, r15, and r18, were set as edge nodes. Then r3, r4, r5, r8, r9, r17, and r19 were individually added as edge nodes. The dotted ovals in Figure 5 denote the combinations of an "exporter" and "importer" for only EAR, where an "importer", denoted in black, uses the SPT of an "exporter", denoted in red.

In this evaluation, three types of links in the network, each of which had either 1,000, 2,000, or 4,000 units of bidirectional bandwidth, were used. The link noted as "2k" means it had 2,000 bandwidth units, "4k" had 4,000 units, and links with no notation had 1,000 units. It was assumed that there was a 10-bandwidth-unit demand between any two different edge nodes.

Figure 6 shows the number of sleeping nodes, and Figure 7 shows the number of sleeping links after each algorithm determined the routes among edge nodes. In these figures, "SPT" indicates the SPT-based algorithm, "Steiner" indicates the proposed Steiner-tree-based algorithm without using bypass routes, and "Steiner(r:2)" indicates the proposed Steiner-tree-based algorithm with bypass routes whose "radius" was set to 2. EAR could not be applied when there were seven edge nodes because the edge nodes were not next to each other, as shown in Figure 5.

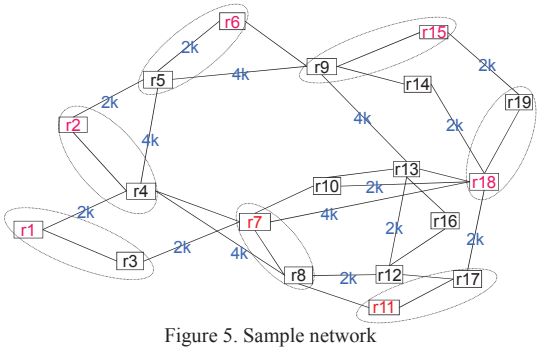

Figure 5. Sample network

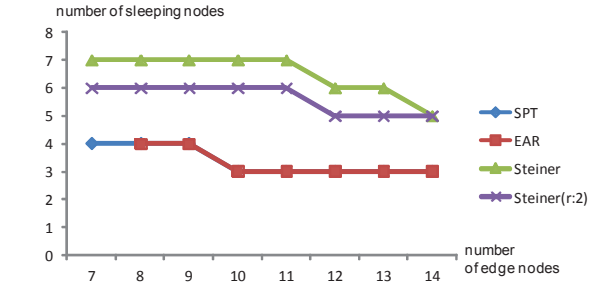

Figure 6. Number of sleeping nodes with each algorithm in sample network

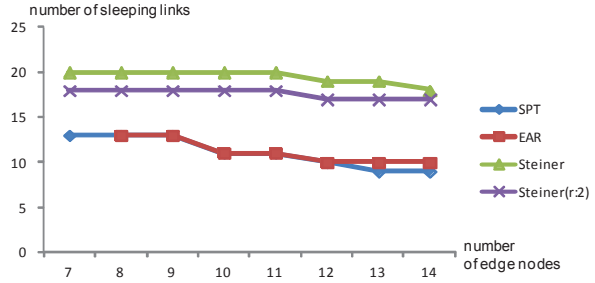

Figure 7. Number of sleeping links with each algorithm in sample network

From Figure 6, we can see that EAR could not increase the number of sleeping nodes compared with "SPT", whereas

"Steiner" could increase the number of sleeping nodes by up to 4. "Steiner(r:2)" was also superior to "SPT" and EAR by up to 3 more sleeping nodes; the number of sleeping nodes was equal or one node smaller than that of "Steiner".

From Figure 7, "Steiner" increased the number of sleeping links by up to 10, compared with "SPT". "Steiner(r:2)" also increased the number of sleeping links by up to 8. On the other hand, EAR barely increased the sleeping links compared with "SPT".

Figure 8 shows the average hop-counts of the routes among all the edge nodes after each algorithm created the routes. "Steiner" had the longest hop-counts, whereas "SPT" had the shortest. When there were seven edge nodes, "Steiner" was about 1.5 hops longer than "SPT". "Steiner(r:2)", however, shortened the average hop-counts compared with "Steiner", and in all the edge node patterns, the average hop-count gap between "Steiner(r:2)" and "SPT" was less than 1.

Figure 9 shows the max bandwidth in a link of the sample network after each algorithm established the routes among all the edge nodes. It is clear that "Steiner" had to use some links more times because there are less links on the Steiner tree. "Steiner(r:2)", however, reduced the max bandwidth to almost the same level as those of "SPT". They were even smaller than those of EAR in most cases. These phenomena are analyzed in the next subsection.

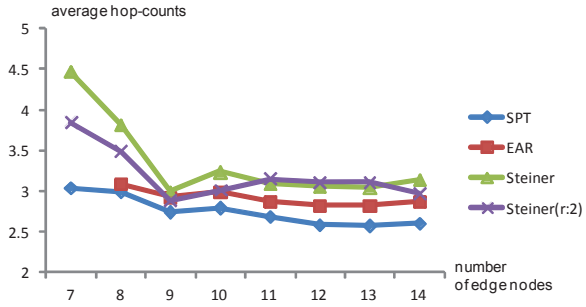

Figure 8. Average hop-counts with each algorithm in sample network

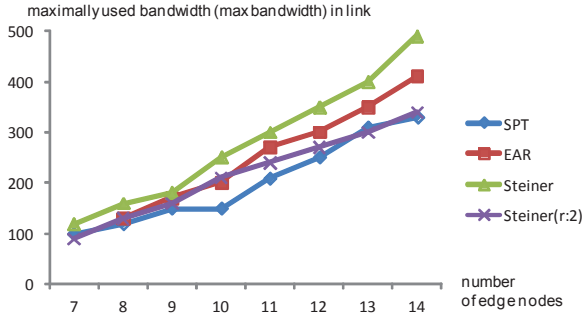

Figure 9. Max bandwidth with each algorithm in sample network

B. *Analyses from evaluation in sample network*

It is clear from these evaluation results that using EAR is not so effective in increasing the numbers of sleeping nodes and links in this small network. On the other hand, a Steiner tree is very effective. "Steiner" increased the number of sleeping nodes and links drastically compared with "SPT". In addition, it is also clear that using bypass routes on the Steiner tree was very effective in reducing the max bandwidth in a link of the network.

Figure 10 shows the links on the sample network used with "SPT" and "Steiner(r:2)". The used links in the network are in either red or blue. "SPT" uses links following the rule that the shortest path between two nodes are selected and does not reduce the numbers of used nodes and links on the entire network. In addition, if there are two routes between two nodes, both routes may be used for the different directions. For example, between r1 and r7 there are two routes: r1-r4-r7 and r1-r3-r7, and their costs are the same. "SPT" used "r1-r4-r7" from r1 to r7 and used "r7-r3-r1" for the opposite direction. For these reasons, the numbers of sleeping nodes and links were much smaller than the case in which a Steiner tree, which is nearly optimized to have the minimum tree cost, was used.

As shown in Figure 10, the "*base_bypass*" was established with "Steiner(r:2)" using the route of r15-r19-r18, which is shown in blue. The edge nodes, which are within 2 hops from r15 or r18, took new bypass routes using this "*base_bypass*" for reaching other edge nodes, which were over the "*base_bypass*" and within the "radius", so congestion on the Steiner tree was abated.

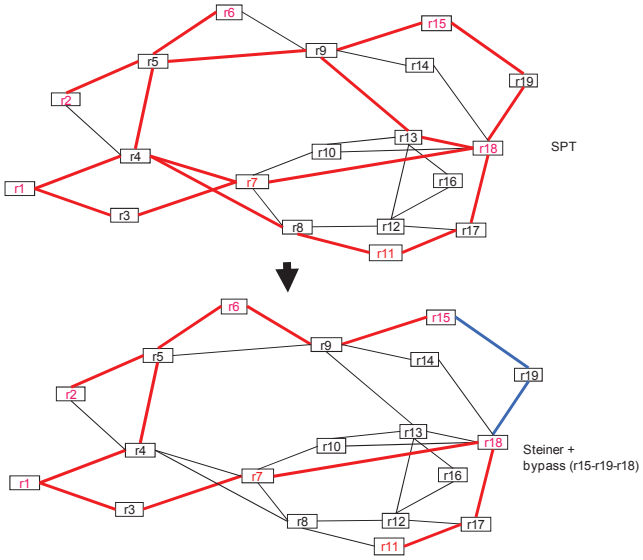

Figure 10. Link usage in sample network with "SPT" and "Steiner(r:2)"

The "radius" used for "Steiner(r:2)" was 2 because, compared with other integers, 2 was more efficient. Figure 11 compares three different "radiuses" in terms of average hop-counts and max bandwidth when there were seven edge nodes, as shown in Figure 10. In both average hop-count and max bandwidth, when the "radius" was set to 2, the algorithm performed the best (Figure 11). This phenomenon is also applicable to other edge node numbers. This is because, in this small network, if the "radius" is set to 3 or larger, the "*base_bypass*" will be congested and there are many relatively longer hop routes that pass through the "*base_bypass*". On the other hand, a "radius" of 1 was too small to abate congested traffic on the Steiner tree.

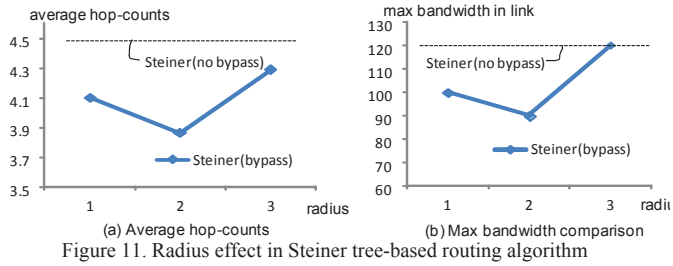

Figure 11. Radius effect in Steiner tree-based routing algorithm

C. *Evaluation in GEANT network*

Figure 12 shows the GEANT network used for the evaluation; it has 40 nodes and 58 links. The group of nodes surrounded by a dotted oval is a group in which an "exporter", denoted in red, exports its SPT to "importers" in the group. A node that has a pair of parentheses after the node name means that the node imports the SPT from the node in the parentheses. These imports were only used with EAR.

In this evaluation, edge nodes increased by 10, which means r1 to r10 were first set as end nodes. Next, r1 to r20 were set then r1 to r30 were set. Finally all the nodes from r1 to r40 were set. The number of nodes was determined in ascending order of their node degrees. This means that r1 had the same or smaller node degree compared with other nodes, and r40 had the largest node degree among all the nodes.

It was assumed there was 1-Mbps traffic demand between two edge nodes if one of the edge nodes had a 155-Mbps interface. It was also assumed that there was 50-Mbps traffic demand between two edge nodes if both edge nodes had a 10-Gbps or larger interface. Otherwise, 10-Mbps traffic demand was assumed between two edge nodes.

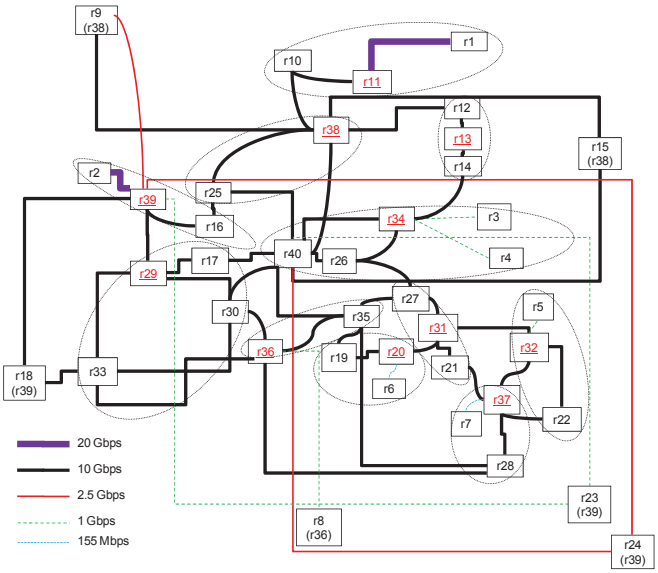

Figure 12. GEANT network used for evaluation.

Figure 13 shows the numbers of sleeping nodes and links after establishing routes among all the edge nodes by using each algorithm. EAR could not be applied to the case with 10 edge nodes because there was no "exporter" among r1 to r10. There was no bypass on the Steiner tree when there were 10 or 20 edge nodes; thus, "Steiner(r:4)", in which the "radius" was set to 4, was applicable from 30 edge nodes. Integer 4 was the most effective "radius" in terms of max bandwidth effect, so it was chosen.

From these results, "Steiner" and "Steiner(r:4)" increased the numbers of sleeping nodes and links at least two times compared with "SPT" and EAR. EAR could not increase the numbers of sleeping nodes and links because a relatively small number of links on the GEANT network did not use the imports of SPTs from the "exporters". That is, even without using the imported SPTs, routes among the edge nodes did not change much.

Figure 14 shows the effects of hop-count and max bandwidth with each algorithm on the GEANT network. The average hop-counts of the routes created using "Steiner" or "Steiner(r:4)" among edge nodes were larger. With 30 edge nodes, there were no good effects (no smaller hop-counts or max bandwidth) by using bypass routes. This was due to the small number of links on the GEANT network, so there was no appropriate "*base_bypass*" that substituted a long hop-count route between two edge nodes.

However, when it comes to 40 edge nodes, the max bandwidth in the network was reduced dramatically from considerably above 10 Gbps, which was the upper limit of the link capacity, to much less than 10 Gbps. It became as small as that of EAR, as shown in Figure 14. This phenomenon was due to the fact that more edge nodes with higher node degrees were added among r31 to r40, so "*base_bypass*" was created between two edge nodes, which had a long hop-count route between them. In this case, a large amount of traffic on the links of the long hop-count route was moved to bypass routes, which used the "*base_bypass*".

Even though a "radius" of 2 was the most effective on the sample network, a "radius" of 4 was the most effective on the GEANT network. This is because GEANT had almost 2 times more nodes than the sample network, so a Steiner tree had about 2 times more links on it.

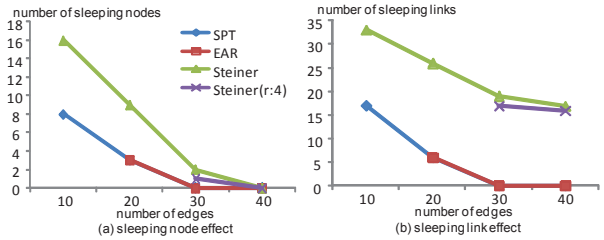

Figure 13. Numbers of sleeping nodes and links in GEANT network

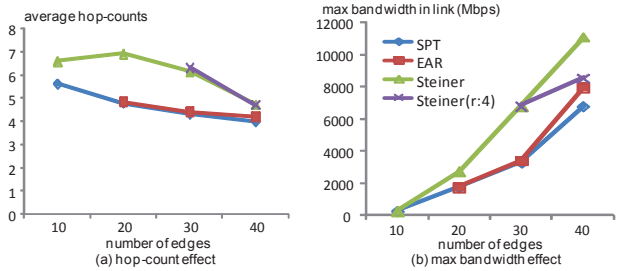

Figure 14. Effects of hop-count and max bandwidth in GEANT network

D. *Evaluation on randomly created network by using Waxman's model*

A randomly created network following Waxman's model was used in this evaluation. The number of nodes in the network was 104 and the number of links was 152, and each link had a different capacity for each direction, upstream and downstream, and each link's capacity was randomly set between 1000 and 2000 units. The average node degree in the network was 2.92 and the maximum node degree was 7. The number of edge nodes increased from 10 to 70 in 20-node intervals. In the evaluation, edge nodes were selected in descending order of their node degrees, so the first ten edge nodes selected had higher node degrees than others. EAR was not applied to this network because it is difficult to determine the "importer"-"exporter" relationships on a randomly created network. Each traffic demand between any two different edge nodes was set to 1 unit.

Figure 15 shows the numbers of sleeping nodes and links after establishing routes among all the edge nodes with each algorithm. The value after the number of edge nodes indicates the "radius" that had the best max bandwidth effect and was used with "Steiner(r)". It is clear that "Steiner" and "Steiner(r)" drastically increased the numbers of sleeping nodes and links compared with "SPT".

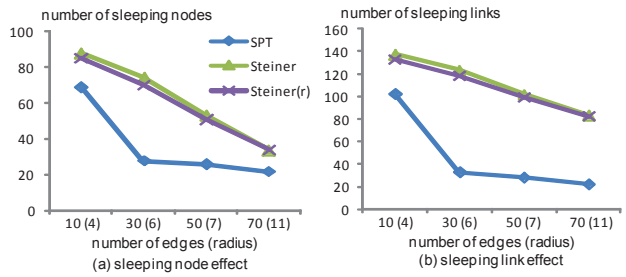

Figure 15. Numbers of sleeping nodes and links in network

Figure 16 shows the hop-count and max bandwidth effects with each algorithm. When the number of edge nodes was small, such as 10 or 30, the hop-count gap between "SPT" and "Steiner"/"Steiner(r)" was about 2 hops, but if the number increased, the gap became wider. In addition, "Steiner(r)" could barely shorten the hop-counts. However, when it comes to the max bandwidth, it is clear that "Steiner(r)" reduced the max bandwidth compared with "Steiner", though the gap between "Steiner(r)" and "SPT" was still large when there were 50 or 70 edge nodes. The "radius" that had the best max bandwidth effect increased when the number of edge nodes increased because the number of links on a Steiner tree increased as edge nodes increased.

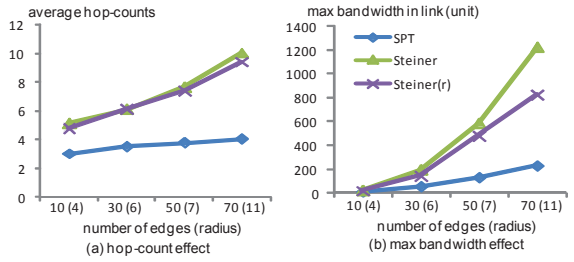

Figure 16. Effects of hop-count and max bandwidth in network

The processing times for calculating the routes among edge nodes with each algorithm were very short, as shown Figure 17. They were all less than 300 ms. The proposed "Steiner" and "Steiner(r)" were faster than "SPT". Even though "Steiner(r)" has the larger time complexities compared with "SPT", as shown in Table 1, the "*base_bypass*" finding process between lines 9 and 15 did not repeat $m(m\text{-}1)$ times and finished in the earlier cycle. However, the larger fluctuation of "Steiner(r)" between 200 and 300 ms depended on how fast "*base_bypass*" was found during the process.

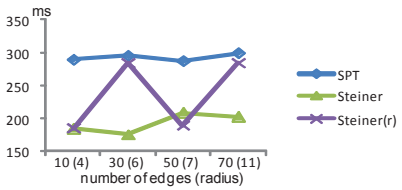

Figure 17. Route calculation time with each algorithm

E. *Real-time network operation applicability of proposed Steiner-tree-based routing*

The amounts of traffic demands among edge nodes fluctuate by time, so unexpected link congestion can sometimes occur. In this case, a real-time response to the congestion is required. From the processing time evaluation, the Steiner-tree-based routing algorithm may be fast enough to be applied to a real-time network operation. However, the proposed bypasses are more useful in the following reasons.

From the evaluation results, in relatively small networks such as the sample and the GEANT, a Steiner tree by using bypasses can increase the numbers of sleep links and nodes significantly with little increase in the max bandwidth. From Figure 17, the processing time gap between Steiner and Steiner(r), which is considered to be the bypass routing time, is also very small.

When a link is congested, a bypass can be set to divert the route, which passes the link, to the bypass route, which does not pass the link, so that congestion is abated. In addition, by adjusting the "radius" from the "*base_bypass*", the network operator can flexibly create bypasses, and other routes unrelated to the bypasses are not affected. Therefore, using bypasses is suitable to a real-time network operation, while maintaining high sleeping rates of nodes and links.

VI. CONCLUSIONS

A Steiner-tree-based energy-saving routing algorithm was proposed, which drastically increases the numbers of sleeping nodes and links in a network. In addition, bypass routes were proposed that replace long inefficient hop-count routes to abate traffic congestion on the Steiner tree.

In the evaluation, the proposed algorithm dramatically increased the numbers of sleeping nodes and links by barely increasing the max bandwidth in a link of a sample and the GEANT networks. On a randomly created network, which had more than 100 nodes, the proposed algorithm further increased the numbers of sleeping nodes and links at the cost of a long hop-count and large max bandwidth. The proposed algorithm also quickly calculated the routes, making it applicable to real-time operations.

REFERENCES

[1] L. Chiaraviglio, M. Mellia, and F. Neri, "Reducing Power Consumption in Backbone Networks," IEEE ICC, 2009.

[2] M. Zhang, C. Yi, B. Liu, and B. Zhang, "Green TE: Power-Aware Traffic Engineering," IEEE ICC, 2010.

[3] A. P. Bianzino, C. Chaudet, F. Larroca, D. Rossi, J.-L. Rougier, "Energy-Aware Routing: a Reality Check," IEEE Globecom Workshop on Green Communications, 2010.

[4] F. Giroire, D. Mazauric, J. Moulierac, and B. Onfroy, "Minimizing Routing Energy Consumption: from Theoretical to Practical Results," IEEE/ACM International Conference on Green Computing and Communications, 2010.

[5] N. Vasic and D. Kostic, "Energy-Aware Traffic Engineering," Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking, pp. 169-178, 2010.

[6] J. Moy, "OSPF Version 2," RFC 2178, Apr. 1998.

[7] E. Mannie (ed.), "Generalized Multi-Protocol Label Switching (GMPLS) Architecture," RFC 3945, Oct. 2004.

[8] J. Wang, S. Ruepp, A. V. Manolova, L. Dittmann, S. Ricciardi, D. Careglio, "Green-Aware Routing in GMPLS Networks," Workshop on Computing, Network and Communications, 2012.

[9] A. Jamakovic and S. Uhlig, "On the relationship between the algebraic connectivity and graph's robustness to node and link failures," NGI 2007, pp. 96-102, May 2007.

[10] F. Cuomo, A. Abbagnale, A. Cianfrani, M. Polverini, "Keeping the Connectivity and Saving the Energy in the Internet," IEEE Infocom Workshop on Green Communication and Networking, 2011.

[11] A. Cianfrani, V. Eramo, M. Listanti, and M. Marazza, and E. Vittorini, "An Energy Saving Routing Algorithm for a Green OSPF Protocol," IEEE Infocom, 2010.

[12] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," Numerische Mathematik, 1, pp. 269–271, 1959.

[13] H. Matsuura, "Proposal of New Steiner Tree Algorithm Applied to P2MP Traffic Engineering," IEEE Globecom, 2012.

[14] H. Takahashi and A. Matsuyama, "An Approximate Solution for the Steiner Problem in Graphs," Math. Japonica, pp. 573–577, 1980.

[15] J. Ash and J. L. Le Roux, "Path Computation Element (PCE) Communication Protocol Generic Requirements," RFC4657, Sep. 2006.

[16] http://www.openflow.org/

[17] http://www.jboss.org/

[18] M. L. Fredman and R.E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," Journal of the ACM (JACM), vol. 34, no. 3, pp. 596-615, 1987.

[19] Java Community Process, "JSR-000220 Enterprise JavaBeans 3.0 (Final Release)," http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html

[20] http://dev.mysql.com/

[21] Q. Ma and P. Steenkiste, "On Path Selection for Traffic with Bandwidth Guarantees," IEEE International Conference on Network Protocols, pp. 191-202, Oct. 1997.

[22] http://www.geant.net/Media_Centre/Media_Library/Pages/Maps.aspx

[23] B. M. Waxman, "Routing of Multipoint Connections," IEEE Journal on Selected Areas in Communications, vol. 6, no. 9, Dec. 1988.