SUMMARY There is a well known Steiner tree algorithm called minimum-cost paths heuristic (MPH), which is used for many multicast network operations and is considered a benchmark for other Steiner tree algorithms. MPH’s average case time complexity is $O(m(l + n \log n))$, where $m$ is the number of end nodes, $n$ is the number of nodes, and $l$ is the number of links in the network, because MPH has to run Dijkstra’s algorithm as many times as the number of end nodes. The author recently proposed a Steiner tree algorithm called branch-based multi-cast (BBMC), which produces exactly the same multicast tree as MPH in a constant processing time irrespective of the number of multicast end nodes. However, the theoretical result for the average case time complexity of BBMC was expressed as $O(n \log m(l + n \log n))$ and could not accurately reflect the above experimental result. This paper proves that the average case time complexity of BBMC can be shortened to $O(l + n \log n)$, which is independent of the number of end nodes, when there is an upper limit of the node degree, which is the number of links connected to a node. In addition, a new parameter $\beta$ is applied to BBMC, so that the multicast tree created by BBMC has less links on it. Even though the tree costs increase due to this parameter, the tree cost increase rates are much smaller than the link decrease rates.

key words: Steiner tree, MPH, BBMC, multicast tree

1. Introduction

Because of the growing service demand for multicast network services, such as IPTV and online lectures, fast and effective multicast routing is necessary. Multicast routing servers based on OpenFlow [1] and PCE [2] protocols are now available; thus, the multicast routing module within a routing server is now very important to meet the required routing speed and provide effective multicast routes. A Steiner tree algorithm is one of the candidates to be applied for these routing modules.

The Steiner problem in networks (SPN) is to find a minimum-cost tree spanning a given subset of nodes in a network, and the SPN is proved as a nondeterministic polynomial time (NP)-complete problem; thus, several polynomial-time heuristic Steiner tree algorithms [3]–[8] have been proposed. The minimum-cost path heuristic algorithm (MPH) [3], developed by Takahashi and Matsuyama in 1980, has been widely applied to multicast network operations [9]–[12] because of its good approximation ratio to the minimum tree cost and its simplicity. In addition, MPH was extended to be applied to a directed network, and it was compared with other Steiner tree algorithms as a benchmark [13]. Parallel processing within the algorithm was proposed to shorten the processing time of MPH [14]. As shown above, there are many applications of MPH and there have been many studies on making MPH faster. Therefore, it is very beneficial if we develop a new algorithm, which is much faster than MPH and creates the same multicast tree as MPH does.

The average case time complexity of MPH is $O(m(l + n \log n))$, where $m$ is the number of end nodes, $n$ is the number of nodes, and $l$ is the number of links in the network. This complexity is derived from the fact that MPH calculates the shortest path between the current tree, which is the determined sub-tree to date, and unresolved end nodes, to which a tree branch has not reached, exactly $m$ times. MPH runs Dijkstra’s algorithm [15] to determine each shortest path, and this algorithm’s average case time complexity is $O(l + n \log n)$ using Fibonacci heaps (F-heaps) [16].

The author recently proposed a Steiner tree algorithm named branch-based multi-cast (BBMC), which produces exactly the same multicast tree of MPH in a constant processing time irrespective of the number of multicast end nodes $m$ in the experimental result [17]. Therefore, when $m$ became larger, the processing time of BBMC was less than one tenth that of MPH. In addition, the processing time of BBMC was almost the same as that of another Steiner tree algorithm named destination-driven multi-cast (DDMC) [8], whose average case time complexity is $O(l + n \log n)$. However, the author only proved that BBMC has the average case time complexity of $O(n \log m(l + n \log n))$ [17], which is still dependent on $m$ and does not reflect the experimental result of $m$-independency in processing time. Thus, this paper proves that BBMC has the average case time complexity of $O(l + n \log n)$ on the condition there is an upper limit of the node degree, which is the number of links connected to a node. This proof contributes to convincing users of BBMC that this algorithm is not affected by an increase in multicast end nodes; thus, performance degradation in accordance with an increase in multicast end nodes is highly unlikely.

In addition to this new discovery of the BBMC average case time complexity, this paper also demonstrates that BBMC’s new optional parameter $\beta$ can reduce the number of links on the multicast tree at the expense of a small increase in the tree cost. In a multicast network operation, not only the tree cost but also links used for each multicast tree should be reduced. This is because link-related resources, such as electricity used for the link-terminating ports and an...
MPLS label for each link, can be conserved if the number of links used on a multicast tree is small. That is, if there is no traffic flowing in a link, the ports terminating the link can be set to sleep mode [18]. Once the MPLS protocol is used to establish the multicast tree, conserving MPLS labels is necessary [19].

BBMC with \( \beta \) makes it possible to add a hop-count weight to each multicast branch candidate in the algorithm, so that it can reduce the number of links on a created multicast tree at the expense of a higher tree cost. However, the number of links decreases much more than the increase in the tree cost. Therefore, BBMC with \( \beta \) is effective when a smaller number of composed links is required.

The implemented multicast routing server data structure is shown in Sect. 2. The BBMC algorithm procedure is discussed in Sect. 3, including the proof for the multicast tree identification between BBMC and MPH and the average case time complexity for BBMC. Sect. 4 compares BBMC with MPH in terms of algorithm speed, and the effectiveness of BBMC multicast trees with varied \( \beta \) is also evaluated. Finally, Sect. 5 concludes this paper.

2. Multicast Routing Server Data Structure

For this paper, the multicast routing server, which uses BBMC or MPH to determine a multicast tree, was implemented in a Linux server, and Java-based JBoss [20] and MySQL database (DB) [21] were used as the platform for the server. Figure 1 shows the typical data structure of the routing server, which can be applied to multicast network operation. The DB accesses from the Jboss are seamlessly conducted based on enterprise Java Beans (EJB) [22], which is included in the JBoss platform.

The MySQL DB keeps network information, namely link and node info of the network. Link info corresponds to each link and maintains its upstream and downstream link costs. Node info corresponds to each node and keeps pointers to its connected links. These data are permanently kept in the DB to be repeatedly used by the Steiner tree algorithms.

On the other hand, transient data in node info, such as “branch candidate”, “branch cost”, and “reach”, are used in a memory space for each Steiner tree algorithm in JBoss; thus, they are always null in the DB. When node info or link info is accessed by a Steiner tree algorithm, it is kept in a memory space in JBoss until the algorithm process is over.

“Branch candidate” indicates a Java reference to one of the shortest branches from the current tree to the node to date in the running algorithm; thus, there is at most one “branch candidate” for each node info. The real branch candidates are stored in path queue (\( PQ \)) and “branch candidate” refers to one of the branch candidates in \( PQ \). “Branch cost” indicates the cost of the “branch candidate”. “Reach” indicates if the branch from the source node to the node is already determined: if “yes”, it means the branch to the node is already determined. “Branch cost” is defined in Fig. 1 (*)1, and \( \beta \) is a newly proposed parameter used only for BBMC. Fewer links on a created multicast tree is one of the goals when \( \beta \) is set to a positive number. When \( \beta \) is set to 0, BBMC creates the same tree as MPH does, as proven later.

There are two algorithms: MPH and BBMC, in the Steiner tree algorithm process, and either one is run to create a multicast tree by specifying the source node and multiple end nodes in the network. MPH calls Dijkstra’s algorithm as many times as there are multicast end nodes because MPH determines the shortest branch from each current tree, which means the multicast tree created by the algorithm to date, to the multicast end nodes whose “reach” is still “no”. Dijkstra’s algorithm uses path queue (\( PQ \)) to register a new branch candidate by using the “insert” or “replace” function and to remove the shortest candidate from \( PQ \) by using the “delete-min” function [17]. \( PQ \) has two different forms to hold branch candidates: one is F-heaps and the other is a binary search tree, so either one is chosen. The determined branches are stored in a group \( R \) and compose the final multicast tree.

In \( PQ \), there is at most one branch candidate corresponding to each node in the network and each branch candidate has its branch cost. If there are multiple branch candidates that have the same end node, one of the branch candidates with the smallest branch cost is chosen as the remaining branch candidate in \( PQ \). That is, if there is no branch candidate that ends with a certain node, the “insert” function is used to insert a new branch candidate, which ends with the node, into \( PQ \). If there is already a branch candidate in \( PQ \) that ends with a certain node, a new branch candidate is compared with the existing branch candidate and the existing branch candidate is replaced with the new branch candidate by using the “replace” function only when the new branch candidate cost is smaller than that of the existing branch candidate. After a branch candidate is removed from \( PQ \) by using the “delete-min” function, a new path candidate with the same end node can be inserted to \( PQ \) by the “insert” function.

BBMC does not call Dijkstra’s algorithm; thus, it directly accesses \( PQ \) to determine the shortest branch from...
the current tree, and the determined branches are stored in \( R \). BBMC with a positive \( \beta \) has to use a binary search tree rather than F-heaps because the F-heaps function, namely “decreaseKey” [16], which corresponds to the “replace” function, requires that the new branch candidate, which replaces the old branch candidate, has to have a smaller real branch cost than that of the old branch candidate. However, this is not necessarily true when \( \beta \) is set to a positive number because the \( \beta \)-weighted branch cost is used when a new branch candidate is compared with the old branch candidate. The BBMC with \( \beta = 0 \) can use F-heaps because only a new branch candidate, whose real branch cost is smaller than the “branch cost” in the node info, replaces the old branch candidate. The determined branches are stored in \( R \), as is the case with MPH.

3. BBMC Algorithm

In this section, the concept of the BBMC algorithm, its procedure and an example, proof for BBMC producing the same tree as MPH, and its average case time complexity, are discussed.

3.1 BBMC Algorithm’s Concept

BBMC initializes branch candidates from a new branch as shown in Fig. 2. In this figure, \( tn1 \) to \( tn4 \) indicate nodes on a current tree and belong to \( U \). \( bn1 \) and \( bn2 \), which is one of the multicast end nodes, are on the new branch from \( tn4 \) to \( bn2 \) and belong to \( V \). \( in1 \) to \( in4 \) indicate independent nodes that do not belong to \( U \) nor \( V \). It is assumed that each link cost is 1 regardless of its direction.

In BBMC, new branch candidates start from a node in \( V \), namely \( bn1 \) or \( bn2 \), and end with the node one-hop away, namely \( in1 \) or \( in3 \), respectively. These two branch candidates are compared with candidates that are already registered in \( PQ \) and have the same branch end nodes, namely \( in1 \) and \( in3 \). In this example, \( tn4-bn1-in1 \) is compared with \( bn1-in1 \) and \( tn2-in2-in3 \) is compared with \( bn2-in3 \). In both cases, the costs of the new branch candidates are smaller than the compared current candidates; thus, these current candidates are replaced with the new candidates.

The shortest branch candidate that happens to have a multicast end node at the end of its branch in \( PQ \) is selected as the next branch, which is added to the new current tree \( (U \cup V) \). Branch candidates starting from the nodes on the new branch are compared with candidates in \( PQ \) as explained above. This cycle is continued until all the branches of the multicast tree are determined.

This branch-based initiation of branch candidates makes it possible to seek a smaller Steiner tree compared with other Steiner tree algorithms. For example, BBMC’s Steiner tree costs were 10 to 21% smaller than those produced by DDMC, which is another Steiner tree algorithm, in the experimental result [17]. DDMC is similar to BBMC except that DDMC only initializes a branch candidate from a new branch destination, namely \( bn2 \) in Fig. 2, whereas BBMC initializes all the nodes on a new branch. Unless all the branch candidates from a new branch are initialized under a fair manner like BBMC does, a Steiner tree algorithm cannot produce a near-optimum tree. BBMC has almost the same algorithm processing time as DDMC, which has the average case time complexity \( O(1 + n \log n) \), so this paper proves BBMC has the same average case time complexity of DDMC when each node degree in the network has a constant value.

3.2 BBMC Algorithm’s Procedure

Figure 3 shows the BBMC pseudo code. BBMC has an input consisting of a source node \( s \), multicast end node group \( T \), and \( \beta \) as an option, which has a positive value. BBMC returns \( R \), which is the group of branches comprising the multicast tree, as its output.

Table 1 lists the functions of \( PQ \) used by BBMC. Though, “insert” and “replace” functions do not have return values, “delete-min” function returns the shortest branch candidate in \( PQ \) after it is removed from \( PQ \).

At line 1, \( s \) is entered into group \( V \), \( T \) is copied to \( E \), and \( R \) is set to \( \{\} \), which means an empty group. At line 2, “reach(\( s \))” is set to “yes” because \( s \) is the first node included in the multicast tree. At line 3, node info for \( v \) except for \( s \) is set. “cand(\( v \))” refers the shortest branch candidate to date to \( v \), “cost(\( v \))” means the cost of “cand(\( v \))”, and “reach(\( v \))” is set to “no” because BBMC has yet to reach it. Specifically, “cand(\( v \))” is a Java reference to the branch candidate to \( v \), which is located in \( PQ \), and if it is set to null, there is no branch candidate to \( v \) in \( PQ \). At line 4, the branch candidate “branch” and the shortest branch from a current tree “shortest_branch” are both initialized as null.

The WHILE loop from lines 5 to 34 is repeated until \( |E| = 0 \), which means BBMC reaches all the multicast end nodes. The FOR loop from lines 6 to 24 is repeated for each node \( b \in V \). The FOR loop from lines 7 to 23 is repeated for each “link(b,c)”, which starts with node \( b \) and ends with node \( c \). From line 8 to 9, “branch” is substituted with “link(b,c)” if “shortest_branch” is null, whereas from line 10 to 11, “link(b,c)” is added to the end of “shortest_branch” and creates a new branch candidate, “branch”. Here “branch” is a branch candidate that starts from a node on the current tree
**Algorithm: BBMC(s, T, β)**

Output: R

1. \( V = \{s\}, E = T, R = () \)
2. FOR every \( v \) \( \in \) \( V \), \( s \) \( \rightarrow \) \( v \) \( \rightarrow \) \( \text{YES} \)
3. FOR every \( (v, w) \) \( \in \) \( E \), \( s \) \( \rightarrow \) \( v \) \( \rightarrow \) \( w \) \( \rightarrow \) \( \text{NO} \)
4. \( \text{branch} = \text{null}, \text{shortest}_\text{branch} = \text{null} \)
5. WHILE \( |E| > 0 \)
6. FOR every \( b \) \( \in \) \( V \)
7. FOR every \( (b, c) \) \( \rightarrow \) \( \text{branch} \)
8. IF [\( \text{shortest}_\text{branch} = \text{null} \)] THEN
9. \( \text{branch} = \text{link}(b, c) \)
10. ELSE THEN
11. \( \text{branch} = \text{shortest}_\text{branch} + \text{link}(b, c) \)
12. IF [\( |\text{branch}| > 0 \) AND \( \text{branch} \in E \) ] THEN
13. \( |\text{shortest}_\text{branch}| = |\text{branch}| + \beta \) branch-hop-count ENDIF
14. IF [\( \text{reach}(\text{branch.end}) = \text{no} \) AND \( (|\text{branch}| < |\text{branch}|) \) < cost(branch.end)]
15. THEN
16. IF [\( \text{cand}(\text{branch.end}) = \text{null} \)] THEN
17. \( \text{PQ.insert(\text{branch.end})} \)
18. ELSE THEN
19. \( \text{PQ.replace(\text{cand}(\text{branch.end}), \text{branch})} \)
20. END IF
21. \( \text{cand}(\text{branch.end}) = \text{branch}, \text{cost}(\text{branch.end}) = |\text{branch}| \)
22. ENDIF
23. END FOR
24. END FOR
25. \( \text{shortest}_\text{branch} = \text{PQ.delete-min} \)
26. IF [\( \text{shortest}_\text{branch.end} \in E \) ] THEN
27. \( R = \text{shortest}_\text{branch} \)
28. IF [\( |R| = m \) ] RETURN ENDIF
29. \( V = \text{on}\_\text{shortest}_\text{branch.end}, E = \text{shortest}_\text{branch.end} \)
30. \( \text{shortest}_\text{branch} = \text{null} \)
31. FOR every \( v \) \( \in \) \( V \), \( \text{reach}(v) = \text{YES} \)
32. ENDIF
33. ELSE \( V = \text{shortest}_\text{branch.end}, \text{cand}(\text{shortest}_\text{branch.end}) = \text{null} \)
34. END WHILE

---

**Fig. 3** BBMC pseudo code.

| \( PQ \) function specifications | \( \text{return} \) | \( \text{behavior} \) |
|----------------------------------|------------------|-------------------|
| insert(\( branch \))            | \( \text{void} \) | insert \( branch \) to \( PQ \) |
| replace(\( branch1 \), \( branch2 \)) | \( \text{void} \) | replace \( branch1 \) with \( branch2 \) in \( PQ \) |
| delete-min                       | \( \text{shortest}_\text{branch} \) | remove shortest branch from \( PQ \) |

**Table 1**

The Java reference to \( \text{branch} \) and \( |\text{branch}| \) is set to \( \text{null} \), which is represented by \( \text{branch.end} \) at line 12.

From lines 12 to 13, \( \beta \) is applied only when \( \beta > 0 \). The branch cost \( |\text{branch}| \) is set to \( |\text{branch}| + \beta \) (branch-hop-count) only when end node of the branch \( \text{branch.end} \) \( \in \) \( E \). In this formula, \( \text{branch-hop-count} \) means the hop-count of \( \text{branch} \).

From lines 14 to 22, \( \text{branch} \) is compared with \( \text{cand}(\text{branch.end}) \), which refers to the shortest branch candidate to the branch end node to date, and if \( |\text{branch}| \) is smaller than \( \text{cost}(\text{branch.end}) \), \( \text{branch} \) is listed to \( PQ \) by using the “insert” or “replace” function at lines 17 and 19, respectively. As shown at line 16, when \( \text{cand}(\text{branch.end}) \) is set to \( \text{null} \), \( PQ \) does not have the branch candidate whose end node is \( \text{branch.end} \). In this case, \( \text{branch} \) is newly inserted to \( PQ \) by using \( PQ \).insert at line 17. Otherwise, the branch candidate referred by \( \text{cand}(\text{branch.end}) \) is replaced with \( \text{branch} \) by using \( PQ \).replace at line 19. After listing \( \text{branch} \) to \( PQ \), \( \text{cand}(\text{branch.end}) \) and \( \text{cost}(\text{branch.end}) \) are replaced with the Java reference to \( \text{branch} \) and \( |\text{branch}| \), respectively, in the node info for \( \text{branch.end} \) at line 21.

At line 25, the shortest branch in \( PQ \) is removed from \( PQ \) as \( \text{shortest}_\text{branch} \) by using \( PQ \).delete-min. Lines 26 to 32 show the procedure when \( \text{shortest}_\text{branch.end} \) \( \in \) \( E \), which means the end node of \( \text{shortest}_\text{branch} \), is one of the multicast end nodes. At line 27, \( \text{shortest}_\text{branch} \) is entered into \( R \). At line 28, the algorithm checks if \( |R| \) has \( m \) branches, where \( m \) is the number of end nodes of the multicast. If there are already \( m \) branches in \( R \), \( R \) is returned as the output of BBMC. At line 29, all the nodes on \( \text{shortest}_\text{branch} \), except the start node of \( \text{shortest}_\text{branch} \), replace all the nodes in \( V \), and the node \( \text{shortest}_\text{branch.end} \) is removed from \( E \). At line 30, \( \text{shortest}_\text{branch} \) is set to \( \text{null} \), so that in the next WHILE loop every \( \text{branch} \) is replaced with \( \text{link}(b, c) \) at line 9. This procedure implies that \( \text{branch} \) selected at line 9 starts with a node that belongs to \( V \). At line 31, \( \text{reach}(v) \) is set to “YES” for each node \( v \) \( \in \) \( V \) because it is newly added to the current tree.

The procedure when \( \text{shortest}_\text{branch.end} \) \( \notin \) \( E \) is shown at line 33. In this case, \( V \) is replaced with only \( \text{shortest}_\text{branch.end} \), and \( \text{cand}(\text{shortest}_\text{branch.end}) \) is set to \( \text{null} \). This is because there is no branch candidate whose end node is \( \text{shortest}_\text{branch.end} \) in \( PQ \).

3.3 BBMC Procedure Example

Figure 4 shows an example how BBMC computes a multicast tree in a network. As a condition, \( \beta \) is set to 0 so that real branch cost without using hop-count weight can be considered. In the figure, \( s \) is the source node of a requested multicast tree and \( e1, e2, \) and \( e3 \) are end nodes of the multicast tree. \( n1 \) to \( n4 \) are nodes in this sample network, which are not source or end nodes of the multicast tree. The number besides a directed link indicates the link cost for the link.

In Fig. 4, (1) shows that BBMC lists three branch candidates, \( s-e1, s-n1, \) and \( s-n2 \), to \( PQ \) in the first WHILE loop. These three branch candidates are links from \( s \) and listed to \( PQ \) by using the “insert” function of \( PQ \) at line 17 in the BBMC pseudo code. At the same time, these branch candidates are substituted to \( \text{cand}(e1), \text{cand}(n1), \) and \( \text{cand}(n2), \)
and their branch costs are substituted to cost(\(e1\)), cost(\(n1\)), and cost(\(n2\)) respectively at line 21. The branch candidate \(s\)-\(e1\) is the shortest in \(PQ\); thus, it is removed from \(PQ\) by using the “delete-min” function of \(PQ\), and shortest branch is substituted with \(s\)-\(e1\) at line 25. \(e1\) is a multicast end node, so \(s\)-\(e1\) is determined as one of the determined branches and entered into \(R\) at line 27. The shortest branch is set to null at line 30, and reach(\(e1\)) is set to “yes” at line 31.

In the same figure, (2) shows that BBMC adds one branch candidate, \(e1\)-\(n3\), in the second WHILE loop. In this loop, there is only one node, \(e1\), in \(V\), so two links from \(e1\) are dealt with in the FOR loop from lines 6 to 24. At line 8, shortest branch is set to null, so links from \(e1\) become branch candidates. However, link \(e1\)-\(n1\) is not listed to \(PQ\) because its branch cost [branch] is 5 and larger than \([\text{cand}(n1)]\), which is 4, so that the condition at line 14 prevents the branch candidate from being listed to \(PQ\). Therefore, only the branch \(e1\)-\(n3\) is listed to \(PQ\). The branch candidates \(s\)-\(n1\) and \(e1\)-\(n3\) are the shortest in \(PQ\) because both have the same branch cost: 4. In this case, BBMC arbitrarily selects one of them. In this example, \(s\)-\(n1\) is assumed as selected and removed from \(PQ\) by using the “delete-min” function in line 25, but even if \(e1\)-\(n3\) is selected, BBMC creates the same tree at the end of the algorithm.

In (3), BBMC adds one branch candidate, \(s\)-\(n1\)-\(e2\), in the third WHILE loop. A link from \(n1\) is added to \(s\)-\(n1\) at line 11. However \(s\)-\(n1\)-\(e1\) is excluded because reach(\(e1\)) “has been set to “yes””. \(s\)-\(n1\)-\(n3\) is excluded because its branch cost is 5 and larger than cost(\(n3\)), which is 4. Branch candidate \(s\)-\(n1\)-\(n2\) is also excluded because its branch cost is 7 and larger than cost(\(n2\)), which is 5. Branch candidate \(e1\)-\(n3\) is removed from \(PQ\) and set to shortest branch at line 25 by using the “delete-min” function.

In (4), BBMC adds four new branch candidates in the sixth WHILE loop. In the previous loop, the branch \(s\)-\(n1\)-\(e2\) was added to \(R\); thus, branch candidates from \(n1\) and \(e2\) are compared with existing branch candidates in \(PQ\). Branch candidate \(n1\)-\(n2\) replaces cand(\(n2\)) because its cost is 3, which is smaller than cost(\(n2\)), which is 5. Branch candidate \(e2\)-\(n4\) also replaces cand(\(n4\)), \(n1\)-\(n3\) replaces cand(\(n3\)), and \(e2\)-\(e3\) replaces cand(\(e3\)). Among the four branch candidates, \(n1\)-\(n3\) is the smallest; thus, it is removed from \(PQ\) and set to shortest branch at line 25 by using the “delete-min” function. In the next WHILE loop, \(n1\)-\(n3\)-\(e3\) replaces \(e2\)-\(e3\) because the branch cost is smaller, and \(n1\)-\(n3\)-\(e3\) is entered to \(R\) because \(e3\) is a multicast end node. In this way, BBMC selects the multicast tree at the end of the algorithm.

### 3.4 Proof for Identity between MPH and BBMC Trees

This subsection proves that BBMC creates the same multicast tree as MPH does when \(\beta\) is set to 0 (theorem 1). For this purpose, it is sufficient to prove that BBMC selects the shortest branch from a current tree to node shortest branch end at line 25 in the pseudo code of BBMC (lemma 1), regardless of shortest branch end’s belonging: shortest branch end \(\in E\) or shortest branch end \(\notin E\). This is because MPH selects the shortest branch from a current tree to the multicast end nodes to which the branch has yet to be determined.

(Lemma 1) BBMC selects the shortest branch from a current tree to node shortest branch end at line 25 when \(\beta\) is set to 0.

**Proof:** (step 1) From the first current tree, which consists of \(s\), BBMC conducts the same procedure as Dijkstra’s algorithm until the first branch from \(s\) is entered into \(R\) at line 27. Therefore, every shortest branch selected at line 25 is the shortest branch from \(s\) to shortest branch end.

(step 2) The \(k\)th current tree \(U\) is assumed to satisfy lemma 1 (supposition 1). Under this condition, it will be proven that the \((k + 1)\)th current tree \((U \cup V)\) also satisfies lemma 1.

The branch \((a, b)\) is defined as the shortest branch that BBMC selects at line 25 from the \((k + 1)\)th current tree, and it starts from node \(a\) and ends with node \(b\). In addition, a branch \((x, (x \in U))\) or branch \((y, (y \in V))\) is assumed, and either one has less branch cost than branch \((a, b)\). There are two cases to which node \(a\) belongs: (1) \(a \in U\), and (2) \(a \in V\), as shown in Fig. 5. However, regardless to which case \(a\) belongs, it can be proven that it is contradictory if the cost of either branch \((x, b)\) or branch \((y, b)\) is smaller than that of branch \((a, b)\).

The relationship \(|\text{branch}(x, b)| < |\text{branch}(a, b)|\) is contradictory because from supposition 1, the shortest path from \(U\) to \(b\) has to be listed to \(PQ\). Therefore, if \(|\text{branch}(x, b)|\) is smaller than \(|\text{branch}(a, b)|\), the shortest branch from \(U\) to \(b\) is smaller than \(|\text{branch}(a, b)|\). However, this never occurs because the shortest branch from \(U\) to \(b\) has been listed to \(PQ\); thus, it has to be selected before branch \((a, b)\) by BBMC.

The relationship \(|\text{branch}(y, b)| < |\text{branch}(a, b)|\) is also contradictory because BBMC initializes all the branches from the nodes on \(V\) by setting branch \((a, b)\) as null at line 30. In the next WHILE loop, therefore, branch is substituted with link \((b, c)\) at line 9; thus, all the shortest branch candidates from \(V\) are considered in the following loops and listed
to PQ. Therefore, if there is a branch(y, b) that is shorter than branch(a, b), branch(y, b) has to be listed to PQ; thus, it has to be selected before branch(a, b).

By this reductive absurdum, it can be said that shortest_branch, which is represented by branch(a, b), selected from the (k + 1)th current tree also satisfies lemma 1.

Steps 1 and 2 prove that lemma 1 is true by applying a mathematical inductive method.

(Theorem 1) BBMC produces the same tree as MPH does when β is set to 0.

Proof: From lemma 1, when shortest_branch is selected at line 25, the branch to shortest_branch.end from the current tree is the shortest. From the same current tree, BBMC obtains the shortest branch at line 25 in ascending order of |shortest_branch|. Therefore, the first shortest_branch, in which shortest_branch.end ∈ E, selected at line 25 is the shortest branch from the current tree to E.

In other words, BBMC, as well as MPH, selects the shortest branch from a current tree to the multicast end nodes to which the branch has yet to be determined, and this means BBMC produces the same tree as MPH does.

3.5 BBMC Average Case Time Complexity

In this subsection, the average case time complexities of BBMC are discussed. Based on the data structure, namely F-heaps or a binary search tree in PQ shown in Fig. 1, the average case time complexity is determined.

For both F-heaps and binary search tree cases, the dominant factor of the average case time complexity is how many times BBMC accesses the links at line 7 and the nodes at line 6 in Fig. 3 throughout the algorithm. Figure 6 shows the maximum expected number of accessed links at line 7 from s and each V_i, where V indicates the group of nodes in V updated at line 29 and i indicates the order of the new branch. In other words, if a new branch to one of the multicast end nodes is determined from the ith current tree, the nodes on the new branch except for the start node, belong to V_i. In this figure, α is the maximum node degree in the network, and τ is the number of links, which directly come from nodes in V_i (enclosed with dotted lines in Fig. 6).

The nodes on the ith current tree consists of s and V_1 to V_{i-1}, so determining the maximum expected number of accessed links from the ith current tree is equivalent to determining the sum of the maximum expected numbers of accessed links coming from s and V_1 to V_{i-1}. In other words, they are the last links of branch candidates starting from s and V_1 to V_{i-1}, and these links are added at lines 9 and 11.

As shown in Fig. 6, the number of accessed links at line 7 from each s or V_i(0 < i ≤ m) never exceeds l, which is the number of links in the network. This is because, as proven in the previous subsection, shortest_branch selected at line 25 in Fig. 3 is always the shortest branch from a current tree. V_{i-1} is included in the ith current tree, so if the shortest_branch starts from V_{i-1}, it is the shortest branch from V_{i-1} to shortest_branch.end. In this case, from V_{i-1}, each shortest_branch to shortest_branch.end is selected at line 25 just once from the same current tree, because any branch, which is chosen at line 11 after the selection of the shortest_branch, never satisfies the condition |branch| < cost(branch.end) at line 14 if branch.end is equal to the shortest_branch.end. As only links coming from the end node of a shortest_branch selected at line 25 are accessed using BBMC at line 7, we should consider the ratio in which a shortest_branch comes from V_i.

As i of V_i increases, the ratio in which a shortest_branch comes from V_i, decreases. In Fig. 6, the ratio, in which a shortest_branch comes from V_1 in a current tree, is less than 1. This is because the second current tree consists of s and V_1, so the ratio is less than 1. If i is larger than 1, the ratio, in which a shortest_branch comes from V_1 in the (i + 1)th current tree, is smaller than 1 because there are i different V_i in the (i + 1)th current tree. With the same reasoning, the ratio, in which a shortest_branch comes from V_2 in a current tree, is less than 1/2. This is because the third current tree consists of S, V_1, and V_2, so the ratio is less than 1/2, and if i is larger than 2, the ratio, in which a shortest_branch comes from V_2 in the (i + 1)th current tree, is smaller than 1/2.

Therefore, the ratio, in which a shortest_branch comes from V_i in a current tree, is less than 1/i. It is assumed that each V_i has the same possibility of having the shortest branch to another node because each V_i is expected to have the same number of nodes. This assumption comes from the fact that V_i is the group of nodes on the shortest branch from the ith current tree to the multicast end nodes to which the branch has yet to be determined. Thus, there is no special current tree that has a higher possibility of having more or less hop-counts on the shortest branch compared with other current trees. Though there may be some hop-count differences among the shortest branches from different current trees depending on the network topologies and the multicast end node patterns, they do not depend on the BBMC algorithm procedure. Therefore, the expected hop-count of the shortest branches is the same.
The maximum expected number of accessed links (line 7 in Fig. 3) from \( V_i (i \leq 2\alpha - 2) \) is less than \( t_i + t_i((\alpha - 1)/i + ((\alpha - 1)/i)^2 + ((\alpha - 1)/i)^3 + \ldots) \), as shown in Fig. 6. The first term \( t_i \) is the necessary number of accessed links from \( V_i \) because \( t_i \) is the number of links directly coming from \( V_i \). The second term \( t_i((\alpha - 1)/i) \) is the maximum expected number of accessed links that come from nodes one hop away from \( V_i \). This is because there are at most \( \alpha - 1 \) links from one node except for the link coming from \( V_i \), the number of nodes one hop away from \( V_i \) is at most \( t_i \), and the ratio, in which the shortest \( \text{branch}(a, c) \) is coming from \( V_i \), is less than \( 1/i \).

The third term \( t_i((\alpha - 1)/i)^2 \) is the maximum expected number of accessed links that come from the nodes two hops away from \( V_i \). From the maximum expected number of accessed links that come from nodes one hop away from \( V_i \), there are at most \( t_i((\alpha - 1)/i)^2 \) links coming from the nodes two hops away from \( V_i \). However, \( t_i((\alpha - 1)/i)^2 \) has to be multiplied by \( 1/i \), that is, the ratio in which the shortest \( \text{branch}(a, c) \) is coming from \( V_i \).

Using the same theory of the third term, it is clear that the \( j \)-th term \( t_i((\alpha - 1)/i)^j-1 \) indicates the maximum expected number of accessed links that come from nodes \((j - 1)\) hops away from \( V_i \).

From Fig. 6, it is clear that from \( V_{2\alpha-2} \), the maximum expected number of accessed links is less than \( t_{2\alpha-2} + t_{2\alpha-2}(1/2 + (1/2)^2 + (1/2)^3 + \ldots) + 2(\alpha-2) \). From the definition of \( t_i, \sum_{m=1}^{\infty} t_i \leq l \). Therefore, the total sum of accessed links from \( V_{2\alpha-2} \) to \( V_m \) is less than \( 2l \). From \( s \) or \( V_1 \) to \( V_{2\alpha-3} \), each number of accessed links is \( I \) or less than \( l \). Therefore, the total maximum expected number of accessed links throughout BBMC is \( l(1 + (2\alpha - 3)) + 2l = 2\alpha l \).

Therefore, the maximum expected number of accessed links at line 7 is \( O(al) \). The ratio between the maximum expected number of accessed nodes at line 6 and the maximum expected number of accessed links at line 7 in Fig. 3 is \( n : l \), so the maximum expected number of accessed nodes at line 6 is \( O(an) \).

If F-heaps are used for \( PQ \), it is proven that if there are \( x \) times “insert”, \( y \) times “decreaseKey”, and \( z \) times “delete-min”, the time complexity of the algorithm is \( O(x + y + z \log x) \) [16]. As mentioned earlier, “decreaseKey” corresponds to the “replace” function at line 15 in Fig. 2. The total number of “insert” and “replace” in BBMC never exceeds the number of total links accesses in BBMC; thus, it is \( O(al) \) average case time complexity. The number of “delete-min” at line 25 never exceeds the number of node accesses at line 6, which has \( O(an) \) average case time complexity. The number of “insert” never exceeds the sum between the number of “delete-min” and \( n \). This is because only node \( \text{branch}.end \), in which \( \text{cand}(\text{branch}.end) = \text{null} \), accepts new \( \text{branch} \) by “insert”, as shown in lines 16 and 17, and it occurs for the node only at the beginning or after \( \text{shortest}.branch \) is deleted by “delete-min” at line 25. Therefore, “insert” takes \( O(an) \) average case time complexity, and the average case time complexity of BBMC with F-heaps is \( O(\alpha(l + n \log(an)) \).

In this paper, it is assumed that the maximum node degree in the network is constant. This limit setting is appropriate for network operation because each network node, such as a router or a switch, has a limited number of ports that terminate links connected to the node. When \( \alpha \) is set to a constant number, the maximum expected number of accessed links at line 7 is \( O(l) \) and the maximum expected number of accessed nodes at line 6 is \( O(n) \), so the average case time complexity of BBMC with F-heaps is \( O(l + n \log(n)) \).

If a binary search tree is used for \( PQ \), BBMC’s average case time complexity is \( O(\log(n)) \) because the total number of “insert” and “replace” is \( O(l) \) average case time complexity and both functions take \( O(\log(n)) \) average case time complexity. The number of “delete-min” at line 25, whose time complexity is \( O(\log(n)) \), takes \( O(n) \) average case time complexity; thus, the “delete-min” process does not affect the average case time complexity of BBMC with a binary search tree.

Table 2 lists the average case time complexities of MPH and BBMC depending on the \( PQ \) data structure when the maximum node degree of a network has a constant number. As shown in the table, BBMC’s average case time complexities are much smaller than MPH’s because MPH has to repeat Dijkstra’s algorithm \( m \) times while BBMC can finish the algorithm with the same average case time complexity of one run of Dijkstra’s algorithm and DDMC.

In the experimental results, BBMC’s Steiner tree costs were 10 to 21% smaller than DDMC’s, though BBMC was as fast as DDMC in terms of algorithm speed [17]. This average case time complexity equality between BBMC and DDMC authenticates these results.

|                      | binary tree | F-heaps          |
|----------------------|-------------|------------------|
| MPH                  | \( O(\log n) \) | \( O(m(l + n \log n)) \) |
| BBMC                 | \( O(\log n) \) | \( O(l + n \log n) \) |

4. Evaluation

Two evaluations are discussed in this section. One is the processing time and \( PQ \) access comparison between MPH and BBMC. The other is BBMC \( \beta \)'s effect on the created multicast trees. The processing time indicates the time with which either BBMC or MPH creates a multicast tree in the routing server.

4.1 Evaluation Conditions

The multicast routing server described in Sect. 2 was used. To demonstrate the generality of these evaluations, Waxman’s model [7], which is used to generate a random network, was used. In this evaluation, 100 points were set on both the \( x \) and \( y \) coordinates, and the probability of a node on \( (x, y) \) was set to 50%, resulting in 5,154 nodes.

A link connecting two nodes was placed with probability proportional to its Euclidean distance, resulting in 28,775
bidirectional links, each of which had different link costs depending on the directions. Each link cost was randomly set between 1 and 100; as a result, the average directed link cost was 43.7.

It is assumed that multicast end nodes are not as congested with multicast traffic compared with the source node. Thus, the node that had the highest node degree in the network, which was 32, was selected as the source node. Multicast end nodes were chosen in ascending order of their node degrees beginning with nodes with one node degree.

4.2 Evaluation of Processing Time and Number of PQ Accesses

In this subsection, the processing times between BBMC and MPH are compared as well as the number of PQ accesses by each algorithm to finish multicast tree creation.

For this evaluation, the BBMC with \( \beta \) was set to 0 so that both BBMC and MPH create the same multicast trees. It was confirmed that all 11 multicast trees created with different numbers of end nodes were exactly the same between the two algorithms. F-heaps were used as the PQ because they make average case time complexities faster than a binary search tree does, as shown in Table 2.

Figure 7 compares (1) the processing time and (2) the number of PQ accesses, which means the sum of the F-heaps function calls: “insert”, “replace”, and “delete-min” in each algorithm between the two algorithms. In each horizontal axis, the number of end nodes is increased by 50 until 300 and then by 300 from 300 to 1800.

It is clear that BBMC was faster and had less PQ accesses than MPH. When the number of end nodes became larger, the number of PQ accesses by MPH was larger, 628 times that by BBMC, when the number of end nodes was 1,800. This was caused by the fact that MPH had to run Dijkstra’s algorithm as many times as the number of end nodes.

On the contrary, BBMC’s processing times were consistent, around 9 seconds, regardless of the number of end nodes. This is because the number of PQ accesses is barely affected by the number of end nodes, as shown in Fig. 7 (2). In addition, this result validates that BBMC’s average case time complexity is independent from the number of end nodes \( m \).

4.3 Parameter \( \beta \) Effect on BBMC Multicast Tree

So far, this paper has described the Steiner algorithm BBMC, in which \( \beta \) is assumed set to 0. In this subsection, the effects of \( \beta \) are evaluated. The \( \beta \)-weighted branch cost is defined as \( \text{real branch cost} + \beta \cdot \text{branch hop-count} \) and taken into account at line 13 in Fig. 3. This \( \beta \)-weighted branch cost \( \text{|branch|} \) makes a branch that has a smaller hop-count more likely satisfy the condition \( |\text{branch}| < \text{cost(branch.end)} \) at line 14 in Fig. 7. As a result, more branch candidates with smaller hop-counts are listed to PQ and selected as one of the final branches in \( R \).

A binary search tree was used for PQ instead of F-heaps in this evaluation due to the following reason. When \( \beta \) is set to a positive number, a branch candidate branch that has a larger real branch cost than cost(branch.end) may be listed to PQ at line 19 by the “replace” function. In this case, the listed branch candidate may have a higher real branch cost than its lower branch candidate in a heap tree in the F-heaps. This is not allowed in the F-heaps and causes the F-heaps to select the wrong minimum branch candidate, which does not have the smallest real branch cost in the F-heaps, at line 25. When it comes to a binary search tree, this discrepancy does not occur because the new branch candidate at line 19 is sorted in the binary search tree in the order of its real branch cost.

Figure 8 shows \( \beta \)’s effect on the number of links and the tree cost of a multicast tree created by BBMC. Multicast trees had 6-9% less links when \( \beta \) was set to 10 compared with those when \( \beta \) was set to 0. As mentioned in the introduction section, a smaller number of links on a multicast tree contributes to reducing the amount of used electricity and number of MPLS labels for the links.

Of course, when \( \beta \) is set to a positive number, tree-cost optimality is negatively affected. However, the link reduction rates were always higher than the tree cost increase rates by 2.5–5.8%, as shown in Fig. 8. This is because a branch with a smaller hop-count generally has a smaller branch cost compared with a branch with a larger hop-count. Therefore,
weighting $\beta$ does not mean that an increase in the multicast tree cost is symmetrical to a smaller hop count.

Figure 9 shows $\beta$’s effect on the algorithm processing time. The processing times of BBMC with a binary search tree were slightly larger than those with F-heaps when $\beta = 0$. This result is thought due to the average case time complexity of BBMC with F-heaps being $O(l + n \log n)$, which is slightly smaller than $O(l \log n)$, which is BBMC’s average case time complexity with a binary search tree.

When $\beta$ was set to 10, however, the processing times of BBMC were slightly shorter than those with F-heaps. This result is thought due to the fact that the number of links required for the created trees were reduced by 6 to 9%, as shown in Fig. 8. As discussed in Sect. 2, the created branches are stored in $R$, and if the hop-counts of these branches are shorter, memory space for $R$ can be saved, which increases the algorithm speed. Therefore, setting a positive $\beta$ does not deteriorate the algorithm speed of BBMC.

5. Conclusion

This paper proved that the proposed Steiner tree algorithm BBMC has the average case time complexity of $O(l+n \log n)$ if the maximum node degree in a network is set as a constant. This average case time complexity is much smaller than that of MPH, which has the average case time complexity of $O(m(l+n \log n))$ in the same condition, though MPH and BBMC create identical multicast trees. This time complexity’s independence from the number of multicast end nodes $m$ indicates BBMC’s constant computation burden regardless of $m$ and it is highly likely that BBMC’s processing time will not be affected by $m$ in various multicast routing operations.

In addition, this paper proposed a new parameter $\beta$ to decrease the number of links on a multicast tree created by BBMC.

In the evaluation section, it was demonstrated that BBMC’s processing time is much smaller than that of MPH, especially when the number of end nodes $m$ is large, because of its independence from the number of end nodes.

It was also shown that BBMC can reduce the number of links on the multicast tree by setting a positive number to $\beta$ at a much smaller tree cost increase rate compared with the link decrease rate.

References

[1] http://www.openflow.org/
[2] J. Ash and J.L. Roux, “Path computation element (PCE) communication protocol generic requirements,” RFC 4657, Sept. 2006.
[3] H. Takahashi and A. Matsuyama, “An approximate solution for the steiner problem in graphs,” Math. Japonica, pp.573–577, 1980.
[4] L. Kou, G. Markowsky, and L. Berman, “A fast algorithm for Steiner trees,” Acta Informatica, vol.15, pp.141–145, 1981.
[5] V.J. Rayward-Smith, “The computation of nearly minimal Steiner trees in graphs,” Int. J. Math. Educ. Sci. Technol., vol.14, pp.15–23, 1983.
[6] V.J. Rayward-Smith and A. Clare, “On finding Steiner vertices,” Netw., vol.16, pp.283–294, 1986.
[7] B.M. Waxman, “Routing of multipoint connections,” IEEE J. Sel. Areas Commun., vol.6, no.9, pp.1617–1622, Dec. 1988.
[8] A. Shaikh and K. Shin, “Destination-driven routing for low-cost multicast,” IEEE J. Sel. Areas Commun., vol.15, no.3, pp.373–381, April 1997.
[9] S. Aggarwal, M. Limaye, A. Netravali, and K. Sabnani, “Constrained diameter Steiner trees for multicast conferences in overlay network,” IEEE QSHINE, 2004.
[10] M. Chaitou and J.-L.L. Roux, “On optimizing leaf initiated point to multi point trees in MPLS,” IEEE Fourth International Conference on Networking and Services, 2008.
[11] J. Rugeli, “Distributed multicast routing mechanism for global Point-to-Point networks,” Proc. 20th EUROMICRO Conf., pp.389–395, 1994.
[12] P.-J. Wan, G. Calinescu, and C.-W. Yi, “Minimum-power multicast routing in static ad hoc wireless networks,” IEEE/ACM Trans. Netw., vol.12, no.3, pp.507–514, June 2004.
[13] S. Ramanathan, “Multicast tree generation in networks with asymmetric links,” IEEE/ACM Trans. Netw., vol.4, no.4, pp.558–568, Aug. 1996.
[14] J.-S. Park, W.H. Ro, H. Lee, and N. Park, “Parallel algorithms for Steiner tree problem,” IEEE Third International Conference on Convergence and Hybrid Information Technology, 2008.
[15] E.W. Dijkstra, “A note on two problems in connexion with graphs,” Numerische Mathematik, vol.1, pp.269–271, 1959.
[16] M.L. Fredman and R.E. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” J. ACM (JACM), vol.34, no.3, pp.596–615, 1987.
[17] H. Matsuura, “Proposal of new Steiner tree algorithm applied to P2MP traffic engineering,” IEEE Globecom, 2012.
[18] M. Zhang, C. Yi, B. Liu, and B. Zhang, “Green TE: Power-aware traffic engineering,” IEEE ICC, 2010.
[19] Y. Oh, D. Kim, H. Yoon, M. Do, and J. Lee, “Scalable MPLS multicast using label aggregation in Internet broadcasting systems,” IEEE 10th International Conference On Telecommunications (ICT), vol.1, pp.273–280, March 2003.
[20] http://www.jboss.org/
[21] http://dev.mysql.com/
[22] Java Community Process, “JSR-000220 Enterprise JavaBeans 3.0 (Final Release),” http://jcp.org/aboutJava/communityprocess/final/jsr220/
Hiroshi Matsuura received his B.E. degree from Kyushu University, Fukuoka, Japan, in 1989. He joined NTT Laboratories in 1989, after which he worked on network design and distributed telecommunications management. He joined NTT West Research and Development Center, in 1999, where he was engaged in managing storage area networks and IP network management. He joined NTT Service Integration Laboratories in 2003, and is now engaged in network management and routing architecture for next-generation IP services.