PAPER

Multi-Agent Steiner Tree Algorithm Based on Branch-Based Multicast

Hiroshi MATSUURA^{†a)}, *Member*

SUMMARY The Steiner tree problem is a nondeterministic-polynomial-time-complete problem, so heuristic polynomial-time algorithms have been proposed for finding multicast trees. However, these polynomial-time algorithms' tree-cost optimality rates are not sufficient to obtain effective multicast trees, so intelligence algorithms, such as the genetic algorithm and artificial fish swarm algorithm, were proposed to improve previously proposed polynomial-time algorithms. However, these intelligence algorithms are time-consuming, even though they can reach quasi-optimal multicast trees. This paper proposes the multi-agent branch-based multicast (BBMC) algorithm, which can maintain the fast speed of polynomial-time algorithms while matching the tree-cost optimality of intelligence algorithms. The advantage of the proposed multi-agent BBMC algorithm is its covering of discarded effective branch candidates to seek the optimal multicast tree. By saving these branch candidates, the algorithm incurs tree-costs that are as small as those of intelligence algorithms, and by saving only a limited number of effective candidates, the algorithm is much faster than intelligence algorithms.

key words: Steiner tree, GA, PSO, multicast tree

1. Introduction

There is an increasing demand to apply multicast routing to network operations such as network routings and reroutings, not only to wired networks but also to wireless networks. For these operations, the minimum cost tree can be found by using Steiner tree algorithms [1]–[3]. Here, a Steiner tree's tree cost is defined as the sum of costs of the individual links comprising the tree. The Steiner problem in networks (SPN) is to find a minimum-cost tree spanning a given subset of network nodes (hereafter called nodes), and the SPN is a nondeterministic polynomial time (NP)-hard problem. Thus many polynomial-time heuristic Steiner tree algorithms have been proposed and applied to network operations. However, their output values are sometimes far from optimal, especially if the network is large.

Therefore, population-based intelligence algorithms, such as the genetic algorithm (GA) [4], [5], particle swarm optimization (PSO) [6], and artificial fish swarm algorithm (AFSA) [7], [8], have been recently applied to obtain minimum tree-cost Steiner trees. There are common features among these intelligence algorithms. They have multiple agents, each of which is put on a position, which represents the Steiner tree nodes comprising a Steiner tree, in a search

space. This paper defines a Steiner position as a position on a search space because the minimum tree-cost Steiner tree is sought on the space. They have exploitation and exploration processes. In the exploitation process, a selected agent takes in superior agents' positive characteristics for the lower tree cost by approaching the superior agents in the search space, and in the exploration process a selected agent randomly explores a new Steiner position and replaces itself with the new position if necessary. In the process of each intelligence algorithm, the cost-minimum Steiner position is recorded throughout its process, and the best Steiner position at the end of the algorithm determines the final Steiner tree.

These intelligence algorithms, however, initiate a determined number of agents, which is called the population, randomly on the space that consists of the Steiner positions. Thus, it takes longer to obtain a small-cost Steiner tree if there are many Steiner positions on the space. This is because each intelligence algorithm has to calculate the Steiner tree cost at each Steiner position, and the number of calculations may reach a critical level degrading the network operation performance if the network scale increases. To overcome the slow speed of intelligence algorithms, this paper proposes a multi-agent branch-based multi-cast (BBMC) algorithm (hereafter called multi-agent BBMC) to obtain Steiner trees whose tree costs are as small as those of intelligence algorithms with a speed that is much faster than that of intelligence algorithms.

The BBMC algorithm (hereafter called BBMC) [9] was developed to create exactly the same Steiner tree made by the minimum-cost path heuristic (MPH) [10], which is referred to as a better polynomial-time Steiner tree algorithm than others in terms of tree costs. In BBMC, an old branch candidate for a node is replaced with a new branch candidate for the same node if the new branch candidate's cost is smaller than that of the old branch candidate. However, these discarded branch candidates may be more appropriate for obtaining a smaller Steiner tree rather than the new branch candidates, at the end of the algorithm.

Therefore, the first advantage of multi-agent BBMC is that the discarded branch candidates in BBMC are used by multiple BBMC agents listed in the BBMC agent list, so smaller-cost Steiner trees can be created compared with BBMC.

The second advantage is that multi-agent BBMC has an evaluation process in which it is judged whether a BBMC agent is to be removed from the BBMC agent list on the

Manuscript received October 7, 2015.

Manuscript revised June 22, 2016.

Manuscript publicized August 8, 2016.

[†]The author is with NTT Network Technology Laboratories, NTT Corporation, Musashino-shi, 180–8585 Japan.

a) E-mail: matsuura.hiroshi@lab.ntt.co.jp

DOI: 10.1587/transinf.2015EDP7417

basis of its created tree cost at this evaluation point, the number of unreached end nodes, and so on. In doing so, only BBMC agents that are promising for creating a smaller Steiner tree remain in the agent list; thus, the speed of the multi-agent BBMC is much faster than typical intelligence algorithms.

The third advantage is that multi-agent BBMC never produces the same Steiner tree with different BBMC agents, though it covers all of the possible trees if there are no restrictions made. An intelligence algorithm may search one Steiner position multiple times, thus creating the same Steiner tree in its algorithm process. Therefore, multi-agent BBMC is more efficient than intelligence algorithms.

The forth advantage is that multi-agent BBMC can be applied to directed networks as well as undirected networks, which are targeted by intelligence algorithms. The upstream traffic pattern is now often totally different from the downstream traffic pattern; thus, it is necessary to set a different link-cost depending on its directions.

The rest of this paper is organized as follows. In Sect. 2, related work is discussed. In Sect. 3, the formal problem formulation and related terms are defined. In Sect. 4, the multi-agent BBMC procedure, including its data structure, pseudo code, application example, optimality, and the average case time complexity, is explained. In Sect. 5, the evaluation results both in directed and undirected networks are given for comparing multi-agent BBMC with AFSA, which approaches the minimum tree further than other intelligence algorithms. Finally, the paper is concluded in Sect. 6.

2. Related Work

As a survey paper [11] shows, several polynomial-time heuristic Steiner tree algorithms have been proposed [10], [12]–[14]. The MPH developed by Takahashi and Matsuyama in 1980 has been widely applied to multicast network operations [15]–[18], and there has been a study done to enhance its speed by using parallel algorithm processing [19] due to its relatively good approximation ratio to the minimum tree cost. The MPH was also applied to a directed network and was compared with other Steiner tree algorithms [20]. Therefore, it would be beneficial to develop an algorithm that is much faster than the MPH and creates the same multicast tree.

The BBMC algorithm was proposed to create exactly the same Steiner tree as the MPH does [9]. In that paper, BBMC's average case time complexity was reduced from MPH's $O(m(l + n \log n))$ to $O(\log m(l + n \log n))$, where m is the number of multicast tree end nodes, n is the number of nodes, and l is the number of links in the network. It was also proven that BBMC's average case time complexity decreases to $O(l + n \log n)$ if each node in the network has an upper limit in its node degree [21]. In quantitative analysis [9], the processing time of the BBMC is independent from m , and it is almost the same as that of destination-driven multi-cast [14], which has the average case time complexity of $O(l + n \log n)$.

Intelligence algorithms have recently been developed [4]–[8], and they can create smaller cost Steiner trees at the expense of longer algorithm processing times. Their combinations between the exploitation and exploration processes are different, but the main idea with them is comparing and moving the Steiner positions of individual agents repeatedly until the end of the algorithm. Whenever an algorithm compares the Steiner tree cost of a Steiner position of an agent with the least tree cost, it has to calculate the tree cost, and the total calculation times may become large enough to cause a network operation delay when the target network scale is large. To alleviate this high burden for each algorithm, parallel processing of multiple agents was proposed [5] to shorten the entire processing time of the algorithm. However, it is necessary to increase the number of machines or CPUs allocated to each agent.

The AFSA is superior to other intelligence algorithms in terms of its approximation rate to the optimal Steiner tree. Thus, in the evaluation section, multi-agent BBMC is compared with AFSA.

3. Problem Formulation

In this section, the formal problem formulation tackled in this paper is described. The problem is defined as directed Steiner tree (DST) problem, and DST is applied on a weighted directed network $G = (V, A, c)$, where V is a set of nodes, A is a set of directed links, and c is a function to determine directed link costs $c: A \rightarrow R^+$ (where R^+ denotes the set of positive values).

The directed link between two nodes, u and v , is represented by $\text{link}(u, v)$, and this means the link starts from u and ends with v . The cost of $\text{link}(u, v)$ can be different from that of $\text{link}(v, u)$ in a directed network, whereas they must be equal in an undirected network. The in-degree of a node u is the number of links (v, u) in the network, and the out-degree of u is the number of links (u, v) . A DST for a directed network G is defined as follows.

(Definition 1) Given a network $G = (V, A, c)$, a set of multicast tree end nodes E , and a source node of the tree $s \notin E$, a DST is a subnetwork $T = (V_t, A_t, c)$ of G , where $V_t \subseteq V$, $A_t \subseteq A$, and $E \subset V_t$. There is a directed path in T from s to every end node, in-degree $(s) = 0$, in-degree $(v) = 1$ for every $v \in V_t - \{s\}$, out-degree $(v) \geq 1$ for every $v \in V_t - E$, and out-degree $(v) = 0$ for every $v \in E$.

Now, the DST problem is ready to be defined as follows.

(Definition 2) Given a network $G = (V, A, c)$, a set of multicast tree end nodes E , and a source node of the tree $s \notin E$, the DST problem is to find the DST whose tree cost is the smallest.

The Steiner problem in undirected networks is a special case of this DST problem where the cost of $\text{link}(u, v)$ is equal to that of $\text{link}(v, u)$. Thus, the proposed Steiner tree algorithm in this paper can be applied to undirected networks as well as directed networks.

4. Multi-Agent BBMC

In this section, at first BBMC, which is used in the proposed multi-agent BBMC multiple times, is explained. After that, the data structure, the pseudo code, an application example, the optimality, and the average case time complexity for multi-agent BBMC are discussed.

4.1 BBMC Algorithm

BBMC is a Steiner tree algorithm, in which a newly determined multicast branch (hereafter called “branch”) from a subtree becomes a base for creating the rest of the tree. Here, a subtree T_i is defined as follows.

(Definition 3) T_i is the subtree in the middle of the BBMC algorithm, where i is the number of multicast tree end nodes reached by the subtree, $T_0 = \{s\}$, and $i < |E|$, which is the number of multicast tree end nodes.

Given a set of nodes on T_i : $V_{T(i)}$, a set of multicast tree end nodes reached by T_i : $E_{T(i)}$, a branch(v_s, v_e) starting from node v_s and ending with v_e , a set of nodes on the branch, V_b , a set of directed links on the branch, A_b , the definition of a branch is as follows.

(Definition 4) If branch(v_s, v_e) starts from T_i , then $v_s \in V_{T(i)}$, $v_e \in E - E_{T(i)}$, $v \notin V_{T(i)}$ for every $v \in V_b - \{v_s\}$, $V_{T(i)} \cup V_b = V_{T(i+1)}$, and $A_{T(i)} \cup A_b = A_{T(i+1)}$.

BBMC finds the shortest branch, which means the smallest cost branch, from each subtree like MPH does. MPH, however, has to run Dijkstra’s algorithm [22] as many times as m , which is the number of multicast tree end nodes, to find the m shortest branches from individual subtrees.

In comparison, BBMC does not run any legacy algorithm to find the shortest branch from each subtree; rather, it gains all of the shortest branches from individual subtrees in one run of its algorithm, whose average case time complexity is as small as that of Dijkstra’s algorithm.

BBMC repeatedly chooses the shortest branch candidate from the branch candidate list (BCL) as a possible branch until the multicast tree reaches all of the specified end nodes. The important feature of BBMC is, whenever a new branch is determined from subtree T_i , the branch candidates starting from a node on the branch are added to BCL.

Figure 1 shows an example of BBMC application. In this example, the network consists of 11 nodes: 1 source (s) node, 4 end nodes ($e1-e4$), and 6 non-end nodes ($n1-n6$). In this example, it is supposed that BBMC started from s and has already determined the subtree consisting of $s, n1, e1$, and $e2$, and the subtree topology is now held in R . All of the branch candidates used by BBMC are held in BCL, and the smallest branch candidate in the BCL is chosen by BBMC.

In Fig. 1, $e2-n4-e3$ is the smallest branch candidate in BCL, so it is chosen and removed from BCL by BBMC, and this is determined as a new branch and sent to R because it ends with an end node. In this case, new one-hop branch candidates, $n4-n3$ and $e3-n5$, which start from the new branch, are added to BCL. In each case, there is

Fig. 1 BBMC application example.

a branch candidate ending with the same branch end node, $n3$ or $n5$ in BCL, and the new branch candidates replace these current branch candidates. This is because the new branch candidates have smaller branch costs than do the current branch candidates. If a new branch candidate does not have a smaller branch cost compared with the current branch candidate, it does not replace the current branch candidate, and it is not listed to BCL.

Next, from BCL, $n4-n3$ is chosen as the smallest branch candidate, but it ends with a non-end node, so after being removed from BCL, one-hop-added branch candidates from $n3$ are considered to be added to BCL. In this example, $n4-n3-e4$ is compared with $e1-n3-e4$ because $e1-n3-e4$ is the current branch candidate ending with the same branch end node, $e4$, and if $n4-n3-e4$ has a smaller branch cost, it replaces $e1-n3-e4$ in BCL.

Thus, BBMC creates branch candidates starting from a newly created branch, while BBMC also creates branch candidates by adding one hop to the smallest branch candidate in BCL ending with a non-end node. This BBMC process continues until all of the end nodes are reached and a Steiner tree is determined in R .

4.2 Data Structure for Multi-Agent BBMC

Multi-agent BBMC has multiple BBMC agents, where each BBMC agent runs the BBMC algorithm with different conditions. As shown in Fig. 2, each BBMC agent has its necessary data in memory space. Agent(0) is for the original BBMC algorithm, while agent(i) ($i > 0$) is for a deviated BBMC agent from another BBMC agent, which is called the parent agent for the deviated agent in this paper.

In the figure, the database keeps the permanent network topology data, which are downloaded to the shared memory space for all the BBMC agents when multi-agent BBMC runs and uploaded to the database when multi-agent BBMC ends its process. As the permanent network topology data, link info keeps the link cost for each direction of the link. Node info keeps links connected to the node. On this network, a Steiner tree with specified multicast tree end nodes

Fig. 2 Multi-agent BBMC data structure.

is created by using multi-agent BBMC.

Besides the permanent network topology data, each node info has transient data for each BBMC agent, which are only used during one run of multi-agent BBMC. That is, as shown in the figure, each node info has a pointer to the branch candidate ending with the node, which has the smallest branch cost up to this point, and its branch cost. The “reach” attribute is for determining if the branch to this node is already determined by the BBMC agent: if the branch is determined, “reach” is set to “yes”; otherwise, it is set to “no”.

The branch backup list in a node info is optionally used and newly added for multi-agent BBMC, which was not necessary for BBMC. The branch candidates, whose branch costs are larger than that of the branch candidate ending with the node, are listed to a binary search tree as the backup branch candidates in the order of their branch costs.

For each BBMC agent, the BBMC process accesses the network information such as link info and node info through the node/link access functions shown in Fig. 2. The BBMC process registers the branch candidate to every node into BCL using the Fibonacci-heaps (F-heaps) access functions [23] “insert” and “decrease-key”. The smallest branch candidate in the BCL is set as a determined branch if it has a multicast tree end node, and listed in R (Fig. 2). At the same time, the branch is removed from the BCL by the F-heaps access function “delete-min”.

The agent list is located in the shared memory for all the BBMC agents as shown in Fig. 2. The agent list has pointers to deviated BBMC agents whose number is equal to or less than the determined upper limit “ max_agent ”, and only the deviated agents that have relatively smaller agent costs are listed in the agent list. For example, if there are already “ max_agent ” agents in the agent list, the agent whose agent cost is the highest in the list is compared with that of the potential new agent. Only when the potential new agent’s agent cost is smaller than that of the largest-agent-cost agent, the new agent replaces the largest-agent-cost agent in the list.

The agent list has a binary tree to hold BBMC agents

Algorithm: multi-agent BBMC(s, E, max_agent)
Output: best R among multi-agents

```

1   $V(0) = \{s\}, E(0) = E, R = \{\}, agent\_list = \{\}$ 
2  for  $s$  reach( $s$ ) = yes
3  for each  $v \neq s$  and  $v \neq null, cost(v) = \infty, reach(v) = no$ 
4  branch = null, shortest_branch = null
5  WHILE ( $|E(0)| > 0$ )
6  FOR every agent( $k$ )
7  FOR every  $b \in V(k)$ 
8  FOR every link( $b, c$ )
9  IF (shortest_branch = null) THEN branch = link( $b, c$ )
10 ELSE THEN branch = shortest_branch * link( $b, c$ )
11 IF (reach(branch.end) = no AND  $|branch| < cost(branch.end)$ ) THEN
12 IF (cand(branch.end) = null) THEN F-heaps( $k$ ).insert(branch) ENDIF
13 ELSE F-heaps( $k$ ).decrease-key(cand(branch.end), branch) END ELSE
14 cand(branch.end) = branch, cost(branch.end) =  $|branch|$ 
15 ENDIF
16 renew_backup_list(branch); #This is optional
17 END FOR every link( $b, c$ )
18 END FOR every  $b \in V(k)$ 
19 shortest_branch = F-heaps( $k$ ).min_candidate
20 F-heaps( $k$ ).delete-min()
21 IF (shortest_branch.end  $\in E(k)$ ) THEN
22  $R(k) +=$  shortest_branch,  $V(k) = on\_shortest\_branch$ ,
23  $E(k) -=$  shortest_branch.end, shortest_branch = null
24 for each  $v \in V(k)$  reach( $v$ ) = yes
25 agent_list_renewal(agent( $k$ ), agent(0),  $max\_agent$ );
26 ENDIF (shortest_branch.end  $\in E(k)$ )
27 ELSE  $V(k) =$  shortest_branch.end, go to same agent( $k$ ) again END ELSE
28 END FOR every agent( $k$ )
29 END WHILE ( $|E(0)| > 0$ )

```

Fig. 3 Pseudo code for multi-agent BBMC algorithm.

except for agent(0). The “insert” function enters a new agent in the binary tree, the “remove” function removes the specified agent from the tree, and the “get-largest” function obtains the largest-agent-cost agent in the tree.

4.3 Multi-Agent BBMC Algorithm Procedure

Multi-agent BBMC has multiple BBMC agents run in turn to create a new branch until all of the agents reach all the specified multicast tree end nodes. Figure 3 shows the pseudo code for multi-agent BBMC.

At the beginning of the algorithm, there is only one BBMC agent: agent(0), so at line 1, the memory data for agent(0) is set. The term $V(i)$ means the node set for agent(i) used from line 7, $E(i)$ means the multicast tree end nodes which are not reached by agent(i), $R(i)$ means the determined branches by agent(i) stored in R in Fig. 2, and “agent_list” is the agent list shown in Fig. 2.

From line 2 to line 3, every node info for agent(0) is set. For the source node, the “reach” attribute is set to “yes” at line 2. For each of the other nodes v , cand(v), which means the branch candidate to v , is set to null because there are no branch candidates at the first stage. The branch cost of the branch candidate ending with v is represented by cost(v) and is set to ∞ . The “reach” attribute for v is set to “no” because the branch to v has yet to be determined. At line 4, variables “branch” and “shortest_branch” used from line 9 are initialized as null. The WHILE routine from line 5 to line 29 is repeated as long as there is one or more unreached end nodes in $E(0)$, which is the set of unreached end nodes by agent(0). From line 6 to line 28, each BBMC agent, agent(k), runs

its branch seeking process to one of the multicast tree end nodes. From line 7 to line 17, all the links from the nodes in $V(k)$ are targeted and the branch candidates ending with the links are registered to the BCL if necessary. Line 8 to line 16 shows the procedure on how links from one node b in $V(k)$ are targeted.

At line 9, the BBMC process for agent(k) determines if “shortest_branch” is null, which means agent(k) created a new branch in the last FOR routine thus new branch candidates start from the branch. If so, “branch” is substituted with link(b, c), which means the link that starts from node b ends with node c .

If “shortest_branch” is not null, it means agent(k) inherits previously selected “shortest_branch” as the basis for creating new branch candidates. In this case, “branch” is made by concatenating “shortest_branch” with link(b, c), which is represented by “shortest_branch • link(b, c)” at line 10. At line 11, the end node info of “branch”, which is shown as “branch.end”, is checked, and if its “reach” attribute is set to “no” and cost(branch.end) is larger than the cost of “branch”: |branch|, it goes onto the procedure from line 12 to line 14.

If cand(branch.end) is null at line 12, “branch” is inserted to the BCL by using “insert”. If cand(branch.end) is not null, as shown at line 13, a new “branch” replaces the current branch candidate to the end node with the “decrease-” function of F-heaps.

At line 16, renew_backup_list(branch) function updates the backup branch list by reflecting the new branch candidate: “branch”, if the optional backup list is used in the node. That is, the smallest branch candidate is set as the branch candidate, while the other branch candidates ending with the node are listed to the backup branch list in the order of their branch costs.

At line 19, “shortest_branch” is substituted with the smallest branch candidate in the BCL, and this branch candidate is removed from the list by the F-heaps function “delete-min()” at line 20.

From line 21 to line 26, the procedure when the end node of “shortest_branch” is a multicast tree end node belonging to $E(k)$ is shown. At line 22, “ $R(k) += \text{shortest_branch}$ ” means that a new determined branch, “shortest_branch”, is added to $R(k)$. The term “ $V(k) = \text{on_shortest_branch}$ ” means that all the nodes on “shortest_branch”, except for the starting node, replace the nodes in $V(k)$. At line 23, “ $E(k) -= \text{shortest_branch.end}$ ” means that from $E(k)$ the end node of “shortest_branch” is removed. At line 25, the “agent_list_renewal” function is called to renew the agent list reflecting the agent cost of agent(k), which is detailed in the next subsection.

Line 27 shows the procedure when the end node of “shortest_branch” does not belong to $E(k)$. In this case, $V(k)$ is replaced with the end node of “shortest_branch”, and this algorithm goes to line 7 under the same BBMC agent(k). Line 27 indicates that unless a branch is set to $R(k)$, the algorithm does not move to the next BBMC agent($k + 1$).

```

Function: agent_list_renewal(agent( $k$ ), agent(0), max_agent)
1  eva_cost_1 = func_eva_cost(| $R(k)$ |, | $R(0)$ |, | $E(k)$ |)
2  IF ( $k \neq 0$  AND eva_cost_1 > 0) THEN
3      B-tree.remove(agent( $k$ )) END IF
4  ELSE IF ( $k \neq 0$  AND eva_cost_1 ≤ 0) THEN
5      new_cost = func_agent_cost(| $R(k)$ |, | $R(0)$ |, last_dev)
6      B-tree.remove(agent( $k$ )), B-tree.insert(agent( $k$ ), new_cost) END ELSE IF
7  eva_cost_2 = func_eva_cost(|dev_ $R(k)$ |, | $R(0)$ |, | $E(k)$ |)
8  IF (eva_cost_2 ≤ 0) THEN
9      dev_cost = func_agent_cost(|dev_ $R(k)$ |, | $R(0)$ |, last_dev)
10     IF (|agent_list| < max_agent) THEN
11         dev_agent = deviate(agent( $k$ ))
12         B-tree.insert(dev_agent, dev_cost) ENDIF
13     ELSE IF (|agent_list| = max_agent) THEN
14         largest_agent = B-tree.get_largest()
15         IF (dev_cost < largest_agent.cost) THEN
16             B-tree.remove(largest_agent), dev_agent = deviate(agent( $k$ ))
17             B-tree.insert(dev_agent, dev_cost) ENDIF
18         END ELSE IF (|agent_list| = max_agent)
19     ENDIF (eva_cost_2 ≤ 0)

```

Fig. 4 Pseudo code for agent_list_renewal function.

4.4 Agent Evaluation in Agent List

Agent(k) is evaluated if it remains in the agent list whenever it creates a new branch, and it is also evaluated if an agent that deviates from agent(k) should be created. Basically, agent(k) is compared with other agents in terms of their multicast tree costs at this evaluation point by accessing the agent list in the shared memory in Fig. 2. These procedures are conducted in the “agent_list_renewal” function, which is called at line 25 in Fig. 3.

Figure 4 shows the pseudo code for the “agent_list_renewal” function. At line 1, “eva_cost_1” is determined by the *func_eva_cost*, which is defined as

$$\begin{aligned} \text{func_eva_cost}(|R(k)|, |R(0)|, |E(k)|) \\ = |R(k)| - |R(0)| - w_e |E(k)|, \end{aligned} \quad (1)$$

in which $|R(k)|$ is the multicast tree cost by agent(k) up to this point, $|R(0)|$ is the multicast tree cost by agent(0) up to this point, $|E(k)|$ is the number of unreached multicast tree end nodes belonging to $E(k)$ at this point, and w_e is the weight for $|E(k)|$ that has a positive value.

This “eva_cost_1” is used to determine if agent(k) is removed from the multi-agent BBMC process. When k is 0, it means that this is BBMC without using deviations from other BBMC agents; thus, it is not removed. Otherwise, there is a possibility that agent(k) will be removed when “eva_cost_1” has a positive value.

The meaning of a positive “eva_cost_1” is that $|R(k)|$ is fairly larger than $|R(0)|$ despite the subtracted value of $w_e |E(k)|$ from $|R(k)|$. $|E(k)|$ decreases when the multicast tree end nodes reached by agent(k) increase. This $w_e |E(k)|$ subtraction is introduced in this paper because it is highly possible that if the number of unreached multicast tree end nodes $|E(k)|$ is decreased, there will be few chances for agent(k) to reduce the gap between $|R(k)|$ and $|R(0)|$. In other words, if $|E(k)|$ is large, there are more chances for agent(k) to reduce $|R(k)|$ to less than $|R(0)|$ by the end of the algorithm. As a

new BBMC agent inherits all the determined branches except for the newly determined branch, $|R(k)|$ has exactly the same number of determined branches as those in $|R(0)|$ after agent(k) adds a new branch to $R(k)$. If two different subtrees having different number of branches are compared, the subtree having more branches tends to have higher tree cost. In this sense, in Formula (1), $|R(k)|$ is treated fairly with $|R(0)|$.

At line 5, the agent cost “new_cost” for agent(k) is newly determined based on the *func_agent_cost* function represented by

$$\begin{aligned} & \text{func_agent_cost}(|R(k)|, |R(0)|, \text{last_dev}) \\ &= |R(k)| - |R(0)| - w_{dev} \times \text{last_dev}, \end{aligned} \quad (2)$$

where “last_dev” indicates the number of unreached multi-cast tree end nodes of the parent agent when agent(k) deviated from the parent agent. For example, if the deviation occurred when there were i unreached end nodes, “last_dev” equals i . The weight w_{dev} is set to a positive value for “last_dev”. The “last_dev” is considered because it is assumed that the earlier the deviation is the more effect there is from the deviation.

At line 6, agent(k) is removed from the binary tree in the agent list, but it is relisted to the same binary tree by “insert”, so that the same agent(k) is continuously used with the new agent cost “new_cost”.

From line 7 to line 19, multi-agent BBMC creates another BBMC agent. First, the evaluation cost “eva_cost.2” is determined using Formula (1) at line 7. Instead of using $|R(k)|$ as for “eva_cost.1”, $|dev_R(k)|$ is used. The term “dev_ $R(k)$ ” represents the tree deviated from $R(k)$, which removes the “shortest_branch” from $R(k)$ and instead uses either the smallest backup branch candidate to the end node of “shortest_branch” or the smallest branch candidate in the BCL after “shortest_branch” is removed. The branch that has the smaller cost is selected between them. The next branch for the deviated agent may not be either one of them, so it is just an approximated tree cost for “dev_ $R(k)$ ”.

If “eva_cost.2” is 0 or less, the new agent for “dev_ $R(k)$ ” is considered to be created from line 8. At line 9, the agent cost “dev_cost” is determined by the function “func_agent_cost”, which follows Formula (2). However, as in line 7, $|dev_R(k)|$ is used as the first argument of “func_agent_cost”.

If the number of deviated BBMC agents is less than “max_agent”, a new deviated agent is created at line 11 by the “deviate(agent(k))” function and added to the agent list with its “dev_cost” at line 12.

When “dev_agent” is created by deviate(agent(k)), it copies almost all the memory data for agent(k) to the memory space for “dev_agent”. All the branches except for “shortest_branch” in $R(k)$ are copied to R for “dev_agent”. The node info for each node of agent(k) is also copied to that of “dev_agent”. However the branch candidate in the node info of “dev_agent” on “shortest_branch” is replaced with the smallest backup branch candidate if the backup branch list is used for the node. Then, the “reach” attribute is set to “no” in the node info for the nodes on “shortest_branch”.

This is because, in “dev_agent”, “shortest_branch” selected in agent(k) is considered to be non-existent.

Line 13 to line 18 shows the procedure when the number of deviated agents in the agent list is equal to “max_agent”. In this case, the agent that has the largest agent cost “largest_agent” is compared with “dev_agent” in terms of their agent costs, and if “dev_cost”, which is the agent cost of “dev_agent”, is smaller than the agent cost of “largest_agent”, “largest_agent” is replaced with “dev_agent” at lines 16 and 17.

When a new “dev_agent” is created at either line 11 or line 16, “dev_agent” is added at the end of the series of agents in the FOR routine from line 6 in Fig. 3. In other words, if agent(i) is the last agent up to this point, the new “dev_agent” becomes agent($i + 1$), so that all the BBMC agents have the same number of determined branches when the last agent finishes the routine from line 6 to line 28 in Fig. 3.

4.5 Multi-Agent BBMC Application Example

Figure 5 shows an example of applying multi-agent BBMC to create a Steiner tree. For simplicity, “max_agent” is set to 1 and weights are set as $w_e = 2$, $w_{dev} = 10$. The targeted network, to which multi-agent BBMC is applied, has eight nodes, among which there are three specified multi-cast tree end nodes: $e1$ to $e3$. The source node is s and other nodes are shown as $n1$ to $n4$. There are directed links among

Fig. 5 Multi-agent BBMC application example with “max_agent” = 1.

the nodes, and the number beside each directed link denotes the link cost for it. Figures 5(1)–(4) show the procedure of agent(0), and (1')–(4') show the procedure of agent(1), which is deviated from agent(0). Optional backup lists are used just for multicast tree end nodes, and the number of backup branch candidates in a backup list is set to 1.

In Fig. 5(1), agent(0) starts the BBMC process by targeting links coming from s . There are three directed links from s ; thus, all the links are listed as branch candidates to BCL(0), as shown in the figure. Among the three branch candidates, branch candidate $s-n2$ has the smallest branch cost of 2; thus, it is selected as the “shortest_branch” and removed from BCL(0). As the end node of this “shortest_branch” $n2$ is not a multicast tree end node, $V(0)$ is set as $n2$ at line 27 in Fig. 3, and the algorithm goes on to line 6 under the same agent(0).

In Fig. 5(2), agent(0) selects each link from $n2$, and the link is connected after the “shortest_branch”, $s-n2$, at line 10 in Fig. 3. As the result, there are three branch candidates created: $s-n2-n1$, $s-n2-e2$, and $s-n2-e1$. The branch candidate $s-n2-n1$ is not listed to BCL(0) because its branch cost, which is 6, is larger than that of $s-n1(5)$. If backup branch candidates are retained in the backup list for $n1$, $s-n2-n1$ is listed in the backup list. However, in this example, $s-n2-n1$ is discarded because it is set that, except for the multicast tree end nodes, backup lists are not used. The branch candidate $s-n2-e2$ is listed to BCL(0), and $s-n2-e1$ is also listed to BCL(0) because this is the first branch candidate ending with $e1$. In BCL(0), the branch candidate $s-n2-e1$ is the smallest, so it is selected as the “shortest_branch”, and it is also moved to $R(0)$ because $e1$ is a multicast tree end node.

At this point, from line 25 in Fig. 3, there is a judgment if a new BBMC agent is created. At line 1 in Fig. 4, agent(0) is not evaluated by “eva_cost_1” because it is the original BBMC agent. Therefore, the procedure from line 1 to line 6 is skipped. At line 7, “eva_cost_2” for the deviated agent from agent(0) is considered. In the function $func_eva_cost$, $|dev_R(0)|$ is either the branch cost of the backup candidate ending with $e1$ or $s-n3(5)$, which is the shortest branch candidate in BCL(0) after $s-n2-e1$ is removed. In this example, there is no backup candidate ending with $e1$ at this point, so $s-n3(5)$ is selected and $|dev_R(0)| = 5$. $|R(0)|$ is the branch cost of the first branch in $R(0)$: $s-n2-e1$, so $|R(0)| = 3$. The term $|E(0)|$ is the number of multicast tree end nodes in $E(0)$ after $s-n2-e1$ is removed, which is 2. Therefore, “eva_cost_2” = $5 - 3 - w_e \times 2 = -2$ because w_e is set to 2, and this value satisfies the condition of line 8 in Fig. 4.

The cost of the new agent “dev_cost” is determined using the function $func_agent_cost$ shown in Formula (2). The “last_dev” is 2 because this is the number of end nodes in $E(0)$ when agent(1) is deviated from agent(0). Therefore, “dev_cost” = $5 - 3 - w_{dev} \times 2 = -18$ because w_{dev} is set to 10. At line 10 in Fig. 4, the number of deviated BBMC agents in the agent list is 0 and is smaller than “max_agent”: 1; thus, agent(1) is created at line 11 and is listed to the binary tree in the agent list with its agent cost, -18, at line 12.

In Fig. 5(1'), all the data for agent(1) are created based on the data from agent(0). That is, all the BCL data are copied. From $R(0)$, $s-n2-e1$ is not copied to $R(1)$, as shown in Fig. 5(1'), because the newly created branch by the parent agent is removed.

From here, agent(1) starts its BBMC process. In Fig. 5(1'), it is assumed that $s-n3$ is selected as the “shortest_branch”, because it is one of the smallest branch candidates in BCL(1). In Fig. 5(2'), agent(1) adds the links from $n3$ to the selected “shortest_branch” $s-n3$ and lists them to BCL(1). As shown in the figure, there are two branch candidates via $n3$: $s-n3-e1$ and $s-n3-e3$, so they are added to BCL(1). Among the branch candidates, $s-n1$ has the smallest branch cost, so it is selected as the “shortest_branch”.

In Fig. 5(3'), agent(1) adds the links from $n1$ to the selected “shortest_branch” $s-n1$ and lists them to BCL(1). In the case of agent(0), the node info of $n2$ has branch cost 2, so $s-n1-n2$ is not listed to BCL(0). However, when the node info is copied to that of agent(1), the branch cost is set to ∞ because agent(1) determines branch $s-n2-e1$ does not exist. At the end of Fig. 5(3'), $s-n3-e1$ is one of the smallest branch candidates in BCL(1), so it is removed from BCL(1) and added to $R(1)$.

At this point, there is a judgment if agent(1) remains in the agent list by the agent_list_renewal(agent(1), agent(0), max_agent) function shown in Fig. 4. First, “eva_cost_1” is evaluated at line 1 by $func_eva_cost(|R(1)|, |R(0)|, |E(1)|) = 6 - 3 - w_e \times 2 = -1$. This “eva_cost_1” does not satisfy the condition of line 2, so agent(1) is not removed from the agent list. Instead it satisfies the condition of line 4, so the new agent cost of agent(1) is determined as “new_cost” at line 5. Here, “new_cost” = $func_agent_cost(|R(1)|, |R(0)|, last_dev) = 6 - 3 - w_{dev} \times 2 = -17$. Therefore agent(1) is relisted to the agent list with this “new_cost” at line 6.

From line 7, there is a judgment if a new agent is deviated from agent(1). At line 7, $|dev_R(1)|$ is sought based on either the branch cost of the backup candidate for $e1$, or the $s-n3-e3(6)$, which is the smallest branch candidate in BCL(1) after $s-n3-e1$ is removed. In this case, $s-n3-e3(6)$ is selected because there is no backup candidate ending with $e1$, so $|dev_R(1)|$ is set to 6. The $|R(0)|$ consists of a single branch $s-n2-e1(3)$, so $|R(0)| = 3$. $E(1)$ has two unreached multicast tree end nodes: $e2$ and $e3$; thus, $|E(1)| = 2$, so “eva_cost_2” = $6 - 3 - w_e \times 2 = -1$. This value satisfies the condition of line 8.

At line 9, the cost of “dev_cost” is determined by the function $func_agent_cost$ in Formula (2) with “last_dev” = 2. Therefore, “dev_cost” = $6 - 3 - w_{dev} \times 2 = -17$. The new agent in this case is not created, because the agent cost of agent(1) equals this potential new BBMC agent cost. Because there is no agent after agent(1), multi-agent BBMC goes to the second WHILE routine starting from line 5 in Fig. 3.

In the second WHILE routine, agent(0) starts with Fig. 5(3). In $V(0)$, there are two nodes: $n2$ and $e1$. Thus, all the links from these two nodes are considered for be-

ing listed to BCL(0) from line 8 to line 17 in Fig. 3. In this process, old branch candidates in the BCL are prone to be replaced with new branch candidates from $V(0)$ because the new branch candidates start from $V(0)$ and there is only one hop on the branch candidates, which means their branch costs tend to be smaller than others. To $e2$, there are two new branch candidates from $V(0)$, and $n2-e2(4)$ is smaller than $e1-e2(5)$, so $n2-e2$ is listed to the BCL and $e1-e2(5)$ is listed to the backup branch list in the node info for $e2$. The end node of $n2-e2(4)$ is a multicast tree end node, so it is moved from BCL(0) to $R(0)$.

The following is a judgment if a new agent is deviated from agent(0) from line 7 in Fig. 4. $|dev_R(0)|$ is sought based on either the branch cost of the backup candidate to $e2$: $e1-e2(5)$ or $n2-n1(4)$, which is the smallest branch candidate in BCL(0) after $n2-e2$ is removed. In this case, $n2-n1(4)$ is selected because $n2-n1(4)$ is smaller than $e1-e2(5)$. There is already a determined branch: $s-n2-e1(3)$ before $n2-e2(4)$ is added, so $|dev_R(0)| = 3 + 4 = 7$. Here $|R(0)| = 7$ and $|E(0)| = 1$, so “eva_cost_2” = $7 - 7 - w_e = -2$. This value satisfies the condition at line 8, but “dev_cost” for the new agent is “dev_cost” = $7 - 7 - w_{dev} = -10$. This “dev_cost” of the new agent is not smaller than the agent cost of agent(1): -17 . Therefore, a new agent is not deviated from agent(0) in Fig. 5 (3).

After Fig. 5 (3), the procedure moves to agent(1) in Fig. 5 (4'). In $V(1)$ there are two nodes: $n3$ and $e1$. Thus, all the links from these two nodes are considered listed to BCL(1). From $n3$, $n3-e3(1)$ replaces $s-n3-e3(6)$ in BCL(1) because of the smaller branch cost, and $s-n3-e3$ is listed to the backup branch list in the node info of $e3$. From $e1$, $e1-e2(5)$ replaces $s-n2-e2(6)$ in BCL(1) because of the smaller branch cost, and $s-n2-e2$ is listed to the backup branch list in the node info of $e2$. Within BCL(1), $n3-e3(1)$ is the smallest and ends with a multicast tree end node; thus, $n3-e3$ is moved to $R(1)$ as the second determined branch.

At this point, there is a judgment if agent(1) is removed from the agent list by using “eva_cost_1” at line 1 in Fig. 4. The “eva_cost_1” is sought by $|R(1)| - |R(0)| - w_e|E(1)| = 7 - 7 - 2 \times 1 = -2$. This non-positive “eva_cost_1” prevents agent(1) from being removed from the agent list at line 2. At line 5, “new_cost” = $|R(1)| - |R(0)| - w_{dev} \times last_dev = 7 - 7 - 10 \times 2 = -20$ is calculated based on Formula (2) and agent(1) is relisted with this “new_cost” at line 6. Here “last_dev” = 2, because there were two unreached end nodes: $e2$ and $e3$ when agent(1) was deviated from agent(0).

The third routine of the WHILE routine starts with Fig. 5 (4) when agent(0) adds links from $e2$ to BCL(0). As a result, $e2-n4(4)$ and $e2-e3(5)$ are added. After that, one of the smallest branch candidates $n2-n1(4)$ is removed from BCL(0) as the “shortest_branch”.

Eventually, agent(0) has the final branch set $R(0)$: $s-n2-e1(3)$, $n2-e2(4)$, and $e2-e3(5)$; therefore, its tree cost is 12. On the other hand, agent(1) has $R(1)$: $s-n3-e1(6)$, $n3-e3(1)$, and $e3-e2(1)$, so its tree cost is 8 and much smaller than that of agent(0). Thus, $R(1)$ is returned as the return value of multi-agent BBMC. In this way, deviated agents

may outperform original BBMC agent(0).

4.6 Optimality of Multi-Agent BBMC

In this subsection, lemma 1 and theorem 1 prove that multi-agent BBMC covers all the possible multicast trees if there is no restriction set in the algorithm.

(Lemma 1) For any nonnegative integer k and any $e \in E(k)$, all the branches that reach e from $R(k)$ can be calculated using multi-agent BBMC if all the backup branch candidates to a node are retained in the backup branch list in the node info and all the BBMC agents are listed to the agent list without evaluating each agent.

Proof:

Let $R(k)_i$ be defined as the tree created by agent(k) and has i reached end nodes. In the same way, let $E(k)_i$ be defined as the group of unreached end nodes of tree $R(k)_i$.

(1) If $i = 0$, it means $R(k)_i = \{s\}$ and multi-agent BBMC can cover all the branches from s to e if all the backup branch candidates to a node are retained and all the deviated BBMC agents are accepted until all the possible branches from s to e are calculated by these agents. This is because every branch candidate to a node is retained in the node info until it is chosen as the “shortest_branch” at line 19 in Fig. 3.

(2) When $i = m$ ($m \geq 0$), it is assumed that lemma 1 is true. This means from $R(k)_m$, all the possible branches to $\forall e \in E(k)_m$ can be covered by BBMC agents. (Assumption 1)

In this case, $R(k)_{m+1} = R(k)_m \cup V(k)_m$, where $V(k)_m$ is the set of nodes on the “shortest_branch” added to $R(k)_m$ at line 22 in Fig. 3 by agent(k). All the possible branches to $\forall e \in E(k)_{m+1}$ from $R(k)_m$ can be covered by BBMC agents from Assumption 1. In addition, multi-agent BBMC can cover all the possible branches from $V(k)_m$ to e if all the deviated BBMC agents are accepted until all the possible branches from $V(k)_m$ to e are calculated by these agents.

From (1) and (2), the mathematic induction proves that lemma 1 is true.

(Theorem 1) Multi-agent BBMC can calculate all the possible trees starting with node s and ending with the specified end node set if all the backup branch candidates to each node are retained in the backup branch list in the node info and all the BBMC agents are listed in the agent list.

Proof:

(1) When $i = 1$, from lemma 1, $\{R(k)_i\}$ ($k = 0, 1, 2, \dots$) covers all the branches from s to $\forall e \neq s$. Therefore, multi-agent BBMC creates all the possible branches to one of the end nodes from s .

(2) It is assumed that $\{R(k)_i\}$ ($k = 0, 1, 2, \dots$) covers trees of all the possible branches to $\forall E_i$ that is a set of i different end nodes. (Assumption 1)

From lemma 1, BBMC agents can cover all the possible branches from each $R(m)_i$ ($m \geq 0$) in $\{R(k)_i\}$ ($k = 0, 1, 2, \dots$) to an unreached end node $\forall e \in E(m)_i$. In addition, from Assumption 1, $\{R(k)_i\}$ ($k = 0, 1, 2, \dots$) covers all the possible trees that have i end nodes. Therefore, $\{R(k)_{i+1}\}$ ($k = 0, 1,$

2, ---) is equivalent to all the possible trees that have $i + 1$ branches.

From (1) and (2), the mathematic induction proves that theorem 1 is true.

4.7 Average Case Time Complexity of Multi-Agent BBMC

It was proven that when the maximum node degree of a network is fixed, BBMC has the average case time complexity of $O(l + n \log n)$, where l is the number of links in the network and n is the number of nodes [21]. This average case time complexity is the same as that of Prim's algorithm [24]. In other words, BBMC's average case time complexity is independent from the number of multicast tree end nodes, which is different from the case of MPH, whose average case time complexity is $O(m(l + n \log n))$ in which m is the number of multicast tree end nodes. In network operation, we generally do not change the node degree for each switch, though we change the number of multicast tree end nodes depending on application demands, so BBMC has an advantage in network applications because its average case time complexity is $O(l + n \log n)$.

The only difference between BBMC and a BBMC agent in multi-agent BBMC is that in a BBMC agent, a backup branch list is optionally held in each node info. However, if the maximum number of backup branch candidates is set for each backup branch list, the update of backup branches at line 16 in Fig. 3 has constant time complexity. Therefore, in this case the average case time complexity for a BBMC agent in multi-agent BBMC is $O(l + n \log n)$.

In multi-agent BBMC, the maximum number of BBMC agents in one WHILE routine from line 5 in Fig. 3 is bounded by $(\text{max_agent} + 1)$ including agent(0). At the same time, each agent(k) ($k = 1, 2, \dots, \text{max_agent}$) in the agent list finds one "shortest_branch" in each WHILE routine from the tree $R(k)_i$, which has i reached end nodes. Meanwhile, agent(0) also finds one "shortest_branch" from $R(0)_i$, which also has i reached end nodes in the same WHILE routine. In this case, there is no average case time complexity difference between agent(k) ($k > 0$) and agent(0) for finding one "shortest_branch" because each of them finds the shortest branch from a tree with i end nodes to an unreached end node in the same network using the BBMC algorithm.

Agent(0), which is the original BBMC algorithm, has $O(l + n \log n)$ average case time complexity in one run of multi-agent BBMC. Thus, the average case time complexity for all the "BBMC_processes" in Fig. 2 is $O(\text{max_agent}(l + n \log n))$ because one BBMC agent's time complexity is multiplied by the maximum number of BBMC agents in the agent list.

However, multi-agent BBMC has to copy the memory data to the memory space for a deviated agent from the original agent. The node info for each node has to be copied to the new space and listed to the new BCL, so it takes time complexity of $O(n)$ for one deviation process. Therefore, the total average case time complexity for multi-agent BBMC

$f(n, l)$ is

$$f(n, l) = O(\text{max_agent}(l + n \log n) + \text{deviation_times} \times n), \quad (3)$$

where "deviation_times" means the number of deviations throughout multi-agent BBMC.

An intelligence algorithm has to calculate the minimum spanning tree (MST) costs of selected Steiner positions by using MST creation algorithms such as Prim. In addition, before and after Prim's algorithm there are multiple processes that take $O(n)$ time, such as comparing two Steiner positions and trimming the unnecessary nodes from the created MST. Therefore, if "max_agent" and "deviation_times" are much smaller than the number of runs of Prim's algorithm, the speed of multi-agent BBMC is much faster than those intelligence algorithms.

5. Evaluations

In this section, multi-agent BBMC is compared with one of the intelligence algorithms, AFSA, which, it is claimed, can approach the minimum Steiner tree more closely compared with other intelligence algorithms, such as PSO and GA [8].

5.1 Target Network Used for Evaluation

In this evaluation, it is assumed that each node is placed on a square grid in a computer simulation environment. Thirty points are set at intervals of 1 unit on both x and y coordinates; the probability of a node at the (x, y) coordinate is set to 50%. After creating a set of nodes on the square grid, Waxman's network model [25] is used to generate links among the created set of nodes, because it has been widely used to generate a random graph or a network in SPN research. In this model, a link connecting two nodes u and v is placed with probability

$$P(u, v) = \beta e^{\frac{-d(u,v)}{L\alpha}}, \quad (4)$$

where $d(u, v)$ is the Euclidean distance between u and v , L is the maximum distance between two nodes on this network, and α and β are parameters for determining the following features of this network. If β is increased, the probability of link existence between two nodes increases, so the number of links connected to one node also increases. If α is increased, the ratio of a longer link relative to a shorter link increases.

Link costs are randomly set between 1 and 100 for both directions; differently for a directed network and equally for an undirected network. In addition, by setting the parameters of Waxman's model, the average link lengths are set at different values. The NW_1 network has relatively shorter links, which is more common because two nodes near each other tend to have a link between them, while NW_2 has longer links. An undirected network, NW_3, with shorter links is also used for comparison. Table 1 lists the features of each network.

Table 1 Feature values of each network.

	number of nodes/links	shortest/longest link (units)	average link length (units)	average node degree
directed NW_1	449/1756	1/13.45	4.29	7.82
directed NW_2	455/1806	1/68.96	15.64	7.94
undirected NW_3	478/1999	1/14.42	4.32	8.36

Fig. 6 AFSA flowchart.

5.2 Artificial Fish Swarm Algorithm

Figure 6 shows the flowchart of AFSA, which is used for comparison with the proposed multi-agent BBMC algorithm. The population of artificial fishes (AFs) is set to 20, and in the first judgment box, one of the selected AFs, X determines if there are more than 0 and less than 9 AFs other than X within a radius of 12 from X . The radius is the number of different bits between two different Steiner positions (AFs). The AF population of 20, 9 AFs, radius of 12 in the first judge box, 500 repetitions of preying, and targeting radius of 5 for praying and random moving, are all commonly used parameters [8] for AFSA and used in the same way for this research.

In the “Following” process, finding the most superior AF in the radius of 12 from X , in terms of its Steiner tree cost, is attempted, and if it can find the AF, X moves in the direction of the AF by decreasing the different bits between them. This “Following” process has the same effect as the “Crossover” in GA, and its main function is exploiting the positive characteristics of one of its superiors.

If the “Following” process fails, AFSA conducts the “Preying” process, in which it tries to find a superior AF within a radius of 5 from X , and if it finds a superior AF, X is replaced with the AF and goes on to the next iteration cycle.

The “Preying” process involves attempting to find a superior AF at most 500 times. If the “Preying” process fails to find a superior AF, it goes on to the “Random moving” process in which X moves randomly within a radius of 5 from X .

These “Preying” and “Random moving” processes have the same effect as “Mutation” in GA, in which a random small change of bits is compared. This cycle of processes is conducted by each of the 20 AFs in turn. For each AF, this cycle is repeated 400/600/800/1000/1200/1400 times in this evaluation.

As discussed above, AFSA has a similar approach for obtaining the best Steiner tree value, like GA and other intelligence algorithms, so comparing the performance of multi-agent BBMC with AFSA is reasonable to show the superiority of multi-agent BBMC to these intelligence algorithms.

For calculating the Steiner tree for each AF Steiner position, Prim’s algorithm is used to calculate the MST for the position, and all the unnecessary nodes are pruned from the MST after that. If an AF Steiner position consists of multiple subtrees disconnected each other, these subtrees are connected with a virtual link that has the same cost of the MST spanning all of the nodes as is applied in [8]. This method is applied for undirected networks, but by applying Prim’s algorithm from the multicast source node, it is also applied to directed networks for calculating the Steiner tree for each Steiner position for this research.

5.3 Multi-Agent BBMC Parameters

There are two multi-agent BBMC parameters besides “ max_agent ” determined before the algorithm runs. In Formula (1), w_e determines the weight for $|E(k)|$ and eventually affects the costs of “ $eva_cost.1$ ” at line 1 and “ $eva_cost.2$ ” at line 7 in Fig. 4. If w_e is large, these costs decrease when there are many unreached multicast tree end nodes in $E(k)$, and the multi-agent BBMC agents having these smaller costs are prone to remain in the agent list or selected as a new agent. When there are many multicast tree end nodes, there are generally many unreached end nodes; thus, w_e is set to a smaller value compared with the case of smaller number multicast tree end nodes. w_e is set inverse proportionally against the number of multicast tree end nodes: $w_e = 2$ when there are 25 end nodes, $w_e = 0.67$ when there are 75 end nodes, and $w_e = 0.4$ when there are 125 end nodes.

In Formula (2), a large w_{dev} is set to give an advantage to a BBMC agent with an earlier deviation. For this evaluation, w_{dev} is set to 10 so that a BBMC agent with an early deviation has an advantage of remaining in the agent list.

The optional backup branch lists are used only for the multicast tree end nodes, and the number of backup branch candidates listed in a backup branch list is set to 1. This setting comes from the fact that multiple backup branch candidates for a node rarely affect the output tree costs of multi-agent BBMC.

Fig. 7 Evaluation results on NW_1.

Fig. 8 Evaluation results on NW_2.

5.4 Evaluation Results and Analyses

One HP ProLiant ML110 server [26] is used to run the algorithms on Java VM, and there is no parallel processing used. Figures 7, 8, and 9 show the evaluation results in NW_1, NW_2, and NW_3, respectively, where the value of “*max_agent*” varies between 20 and 100 with 20 intervals. Twenty-five, 75, and 125 multicast tree end nodes are randomly selected and multicast trees are created using multi-agent BBMC and AFSA from the source node, which is selected from one of the highest node-degree nodes. In the figures, “M-BBMC” means multi-agent BBMC, while “*max_agent* = 0” means BBMC is run instead of multi-agent BBMC.

In AFSA, the cycle of “following”-“praying”-“random moving” is iterated between 40 and 140 times with 20 intervals, and the three nearest tree costs to that of multi-agent BBMC are shown in each evaluation result. In these figures, the number in parentheses after AFSA is the number of MST creations in the AFSA process, and the number in parentheses after M-BBMC is the value of “*max_agent*”.

Figure 7 shows the evaluation results in NW_1, which

is a directed network with short links. Multi-agent BBMC, which has more than one BBMC agent, reduces the Steiner tree-costs of BBMC (*max_agent* = 0), though some AFSA results have smaller tree-costs than those of multi-agent BBMC.

However, when the processing times are compared between AFSA and multi-agent BBMC, AFSA’s processing times are incomparably longer than those of multi-agent BBMC. This is due to the huge number of MST creations of AFSA, while multi-agent BBMC creates trees only (“*max_agent*” + 1) times in its process. The processing time gap between AFSA and multi-agent BBMC is larger when the number of multicast tree end nodes is smaller (such as end nodes = 25). This phenomenon might be due to the fact that the smaller the number of end nodes, the more alternative routes there are among the end nodes. Therefore, AFSA has to calculate more MSTs to reach quasi-optimal trees.

For example, Fig. 7 (1) shows AFSA has to calculate 285,112 MST trees to have a smaller multicast tree cost compared with multi-agent BBMC, which only calculates 101 or less than 101 BBMC trees. As is discussed, both MST and BBMC trees have the same average-case time

Fig. 9 Evaluation results on NW_3.

complexity of $O(l + n \log n)$; thus, these processing time differences between AFSA and multi-agent BBMC obviously come from the differences of the tree creation numbers between them.

The evaluation results in NW_2, which is a directed network with long links, are shown in Fig. 8. As well as the case in Fig. 7, multi-agent BBMC creates the smallest-cost trees much faster than AFSA. However, compared with Fig. 7, the processing time gaps between AFSA and multi-agent BBMC are smaller. This phenomenon is assumed to have come from the tendency that a network with longer links makes the tree-cost gaps among the trees with the same end nodes longer, so that AFSA can approach the smallest tree cost with smaller number of tree calculations. On the other hand, if a network has short links, there are so many similar-cost trees, so AFSA has to take a long time to approach the smallest tree.

The evaluation results in NW_3, which is an undirected network with short links, are shown in Fig. 9. In this evaluation, multi-agent BBMC's "max_agent" is fixed as 20, but up to four different source nodes, from which the algorithm starts, are used to obtain a better result. For example, "20 × 2" means that multi-agent BBMC with 20 BBMC-deviated agents is run from two different source nodes sequentially, and takes the better result between the two. This

Fig. 10 Number of deviations in NW_1 and NW_2.

method takes advantage of the feature of an undirected network: a final tree can be created from any source node by using multi-agent BBMC.

As shown in Fig. 9, compared with Figs. 7 and 8, AFSA requires a longer time to match the tree costs of multi-agent BBMC. As shown in Fig. 9 (1), even 400 seconds of running AFSA cannot reach the tree cost of BBMC. This phenomenon is assumed to have come from the tendency that an undirected network has more similar cost trees compared to a directed network; thus, AFSA has to calculate more trees to approach the smallest cost tree, especially when there are less end nodes.

Throughout the evaluations, multi-agent BBMC can create an output tree much faster than AFSA because of the small number of tree creations: ("max_agent" + 1). However we have to evaluate the "deviation_times" in Formula (3) because it also affects the processing times.

The numbers of deviations in NW_1 and NW_2 are shown in Fig. 10. The numbers of deviations within a multi-agent BBMC process are much smaller compared with the numbers of created MSTs by AFSA in the same networks, as shown in Figs. 7 and 8. In addition, each deviation's time complexity is $O(n)$ and smaller compared with the tree creation time complexity: $O(l + n \log n)$. This evaluation result also supports the smaller processing time of multi-agent BBMC against AFSA.

However, we have to consider why these smaller numbers of deviations are gained. This is assumed because of a large w_{dev} , which is set to 10 in this evaluation. From formula (2), this large w_{dev} makes a BBMC agent, which deviated from a parent agent earlier, have a smaller agent cost. In this case, there is less chance for a later deviated agent to have less agent cost than those deviated earlier. This is why the number of deviations in multi-agent BBMC is reduced.

In addition, there is another advantage to having a larger w_{dev} . That is, it can retain BBMC agents deviated from the original BBMC earlier in the agent list. Generally, these deviated agents can have totally different routes compared with the original BBMC; thus, there are more chances for multi-agent BBMC to cover a wide range of different routes to reach the same set of multicast tree end nodes. In reality, there are many cases for multi-agent BBMC with a large w_{dev} to have smaller tree-cost outputs compared with the setting of $w_{dev} = 0$.

With multi-agent BBMC, it is not necessarily true that more agents in the agent list output a smaller-cost

Fig. 11 Example of better result by smaller “*max_agent*”.

tree. Sometimes a smaller “*max_agent*”, such as “*max_agent*” = 60, outputs a smaller-cost tree than an agent with “*max_agent*” = 100, like the case in Fig. 7 (3). Figure 11 shows an example of why this phenomenon occurs. In this case, within the BBMC agents with “*max_agent*” = 100, the 35th ranked agent at the end of the first WHILE routine in Fig. 3 is excluded from the agent list because its agent cost has been ranked 100th in the middle of the 3rd WHILE routine and replaced with a new agent. However, this agent produces the smallest tree among all of the possible agents at the end of the algorithm.

This phenomenon would not occur if “*max_agent*” was set to 60 because in this example lower-ranked agents, whose ranks are lower than 60th at the end of the first WHILE routine, have surpassed the rank of the excluded agent by the end of the third WHILE routine as shown in Fig. 11. That is, the 79th to 81st ranked agents in the first routine have become 41st to 43rd in the second routine and 51st to 53rd in the third routine. Besides them, there are deviated agents from these agents that are also ranked higher than the excluded agent. If there are no agents whose ranks are lower than 60th, there is no chance for the 35th agent to be surpassed by these lower ranked agents and the deviated agents from them. In this way, each agent’s agent-cost is updated by Formula (2) routine by routine and some lower-ranked agents and the deviated agents from them can surpass higher-ranked agents as the routine increases.

5.5 Advantages of Using Multi-Agent BBMC and Future Work

For each Steiner position, other intelligence algorithms must calculate its tree cost from scratch. Sometimes they happen to select the same Steiner position. In multi-agent BBMC, however, each BBMC agent inherits the branches and branch candidates from its parent agent excluding the latest selected branch; thus, each agent has a different set of branch candidates, so useless repetition for the same tree calculation never occurs.

Multi-agent BBMC can cover all the possible trees if there is no restriction added, as proven in Theorem 1. However, it is time-consuming to calculate all the possible trees; thus, in multi-agent BBMC, the tree cost of the original BBMC agent is used as the standard, and only hopeful

deviated agents, whose tree costs are near the standard, are held in the agent list. On the other hand, an intelligence algorithm starts with a randomly selected Steiner position, so if there are similar tree-cost Steiner positions around the selected position, it will take a long time to approach the smallest-cost tree.

As a result, throughout the evaluation results in the previous subsection, multi-agent BBMC could create an output tree in 1.72 seconds on average, whereas AFSA took 121.7 seconds. In addition, multi-agent BBMC could decrease the tree cost of BBMC by 1.8% on average, whereas AFSA could decrease it by 1.1%.

Even though there is no numerical comparison between GA and multi-agent BBMC in this paper, it is demonstrated that AFSA creates a tree with a smaller tree-cost [8] than GA does. In terms of the algorithm speed of GA, there are time-consuming traits in GA that are similar to those in AFSA. That is, first, each Steiner position for an agent is chosen randomly on the search space without any standards, and they may access the same Steiner positions multiple times. Therefore, GA is much slower than that of multi-agent BBMC.

For example, class C networks in the OR library, which have 500 nodes and 625–12,500 links, are used for a GA performance evaluation [5], and the network size of the Class C networks is similar to those networks used in this paper for the evaluation. On these class C networks, GA takes more than 3 minutes to create a tree using parallel processing with two processes on average. This means that, with a single process, which is used in this paper, it is estimated that GA will take more than 5 minutes.

However, in the multi-agent BBMC process, there are still some skipped agents, which eventually produce better results compared with the agents selected by multi-agent BBMC, as shown in Figs. 7–9. For example, AFSA decreased a BBMC tree cost by 8.4% at its best in the evaluation results, while multi-agent BBMC decreased a BBMC tree cost by 6.6% at its best. They are mainly skipped due to the heuristic values determined by *func_eva_cost* (Formula (1)) and *func_agent_cost* (Formula (2)). In particular, agents within the agent list are ranked by their agent costs determined by Formula (2); thus, it is important how to determine the weights in the formula and their values. This paper argues that a large value of w_{dev} can prioritize the earlier deviated agents and reduce the total deviation times throughout multi-agent BBMC. Under this heuristics, however, some agents with smaller final tree costs are lost in the agent list; therefore, multi-agent BBMC cannot necessarily obtain the smallest-cost trees. Thus, there is some room for other heuristics for determining the agent costs to obtain better results from multi-agent BBMC.

6. Conclusion

A new Steiner tree algorithm called multi-agent BBMC was proposed. By saving the effective branch candidates discarded by BBMC, multi-agent BBMC can reduce the

Steiner tree cost with only a small number of tree creations, e.g., less than 100. Therefore, it can drastically reduce the processing time of intelligence algorithms for finding a Steiner tree, despite its same level of Steiner tree costs, compared with intelligence algorithms.

The weight value, which is used to determine the agent costs of BBMC agents, was also analyzed. It was found that w_{dev} can be used to adjust the number of deviations and reduce Steiner tree costs.

References

- [1] A. Iyer, P. Kumar, and V. Mann, "Avalanche: Data Center Multicast using Software Defined Networking," *IEEE Infocom*, pp.1–8, 2014.
- [2] H. Matsuura, "Energy-saving routing algorithm using Steiner tree," *IFIP/IEEE IM2013*, pp.378–386, May 2013.
- [3] D. Zhou and J. Gao, "Maintaining Approximate Minimum Steiner Tree and k-center for Mobile Agents in a Sensor Network," *IEEE Infocom*, pp.1–5, 2010.
- [4] A. Kapsalis, V.J. Rayward-Smith, and G.D. Smith, "Solving the Graphical Steiner Tree Problem Using Genetic Algorithms," *The Journal of the Operational Research Society*, vol.44, no.4, pp.397–406, 1993.
- [5] N.V. Huy and N.D. Nghia, "Solving Graphical Steiner Tree Problem Using Parallel Genetic Algorithm," *IEEE RIVF*, pp.29–35, 2008.
- [6] R. Qu, Y. Xu, J.P. Castro, and D. Landa-Silva, "Particle Swarm Optimization for the Steiner Tree in Graph and Delay-Constrained Multicast Routing Problems," *Journal of Heuristics*, vol.19, no.2, pp.317–342, 2013.
- [7] M. Neshat, G. Sepidnam, M. Sargolzaei, and A.N. Toosi, "Artificial fish swarm algorithm: a survey of the state-of-the-art, hybridization, combinatorial and indicative applications," *Artificial Intelligence Review*, vol.42, no.4, pp.965–997, 2014.
- [8] Q. Liu, T. Odaka, J. Kuroiwa, H. Shirai, and H. Ogura, "An Artificial Fish Swarm Algorithm for the Multicast Routing Problem," *IEICE Trans. Commun.*, vol.E97-B, no.5, pp.996–1011, 2014.
- [9] H. Matsuura, "Proposal of New Steiner Tree Algorithm Applied to P2MP Traffic Engineering," *IEEE Globecom*, pp.2720–2726, Dec. 2012.
- [10] H. Takahashi and A. Matsuyama, "An Approximate Solution for the Steiner Problem in Graphs," *Math. Japonica*, pp.573–577, 1980.
- [11] P. Winter, "Steiner Problem in Networks," *Networks*, vol.17, no.2, pp.129–167, 1987.
- [12] L. Kou, G. Markowsky, and L. Berman, "A Fast Algorithm for Steiner Trees," *Acta Informatica*, vol.15, no.2, pp.141–145, Springer-Verlag, 1981.
- [13] 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.
- [14] 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.
- [15] S. Aggarwal, M. Limaye, A. Netravali, and K. Sabnani, "Constrained Diameter Steiner Trees for Multicast Conferences in Overlay Network," *IEEE QSHINE*, pp.262–271, 2004.
- [16] 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*, pp.53–60, 2008.
- [17] J. Rugelj, "Distributed Multicast Routing Mechanism for Global Point-to-Point Networks," *Proceedings of the 20th EUROMICRO Conference*, pp.389–395, 1994.
- [18] 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.
- [19] J.-S. Park, W.W. Ro, H. Lee, and N. Park, "Parallel Algorithms for Steiner Tree Problem," *IEEE Third International Conference on Convergence and Hybrid Information Technology*, 2008.
- [20] S. Ramanathan, "Multicast Tree Generation in Networks with Asymmetric Links," *IEEE/ACM Trans. Netw.*, vol.4, no.4, pp.558–568, Aug. 1996.
- [21] H. Matsuura, "Improvement of Steiner Tree Algorithm: Branch-Based Multi-Cast," *IEICE Trans. Inf. & Syst.*, vol.E96-D, no.12, pp.2743–2752, Dec. 2013.
- [22] E.W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol.1, no.1, pp.269–271, 1959.
- [23] 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.
- [24] R.C. Prim, "Shortest connection networks and some generalisations," *Bell System Technical Journal*, vol.36, no.6, pp.1389–1401, 1957.
- [25] B.M. Waxman, "Routing of Multipoint Connections," *IEEE J. Sel. Areas Commun.*, vol.6, no.9, pp.1617–1622, Dec. 1988.
- [26] Hewlett Packard Enterprise, HP ProLiant ML110, 1 June 2016 <http://h50146.www5.hp.com/products/old/servers/proliant/ml/ml110/ml110_qu.html>

generation IP services.

Hiroshi Matsuura received the 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 returned NTT Laboratories in 2003, and is now engaged in network management and routing architecture for the next-