Received May 11, 2020, accepted May 24, 2020, date of publication June 2, 2020, date of current version June 29, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.2999421*

# Top-Down Construction of Independent Spanning Trees in Alternating Group Networks

**JIE-FU HUANG[1], SHIH-SHUN KAO[1,4], SUN-YUAN HSIEH[2,3], (Senior Member, IEEE), AND RALF KLASING[4]**

[1]Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan 701, Taiwan
[2]Department of Computer Science and Information Engineering, Institute of Medical Informatics, National Cheng Kung University, Tainan 701, Taiwan
[3]Center for Innovative FinTech Business Models, National Cheng Kung University, Tainan 701, Taiwan
[4]CNRS, LaBRI, Université de Bordeaux, 33405 Talence cedex, France

Corresponding author: Sun-Yuan Hsieh (hsiehsy@mail.ncku.edu.tw)

**ABSTRACT** A set of spanning trees in a graph $G$ is called independent spanning trees (ISTs) if they are rooted at the same vertex $r$, and for each vertex $v(\neq r)$ in $G$, the two paths from $v$ to $r$ in any two trees share no common vertex expect for $v$ and $r$. ISTs can be applied in many research fields, such as fault-tolerant broadcasting and secure message distribution in reliable communication networks. Since Cayley graphs have been widely used to design interconnection networks, constructing ISTs on cayley graphs is worth studying. The alternating group network is a subclass of Cayley graphs, and the approach of constructing ISTs in alternating group networks is still unknown. In this paper, we propose a novel and simple top-down approach for constructing ISTs in alternating group networks. The main ideas of the algorithm are to use induction to develop small trees to big trees, to use a triangle breadth-first search (TBFS) process to create a backbone of an IST, and to use breadth-first search (BFS) process to connect the rest of nodes. Compared to other methods of different interconnection networks in the literature, the uniqueness of our method is that it does not need to determine the parent of one node by any rule, on the contrary others determine that by rules. The time complexity in $n$-dimensional alternating group network $AN_n$ is $O(n^2 \times n!)$, where $n!$ is twice the number of nodes of $AN_n$; hence it is polynomial time. We implement the algorithm in PHP and run cases from $AN_3$ to $AN_{10}$. The results reveal that all spanning trees of $AN_3$ to $AN_{10}$ are independent and that our algorithm is accurate and efficient.

**INDEX TERMS** Independent spanning trees, alternating group networks, triangle breadth-first search, interconnection networks, Cayley graph.

## I. INTRODUCTION

A set of spanning trees in a graph $G$ is called independent spanning trees (ISTs) if they are rooted at the same vertex $r$, and for each vertex $v(\neq r)$ in $G$, the two paths from $v$ to $r$ in any two trees share no common vertex expect for $v$ and $r$, namely that they are internally disjoint paths. The IST problem is appealing and has attracted considerable attention. It can be applied in many research fields, such as fault-tolerant broadcasting and secure message distribution [1], [2]. These applications are briefly described below [3]:

- In fault-tolerant broadcasting, assume that there exists $k$ ISTs rooted at node $r$ in a network $M$, and $M$ contains at most $k-1$ faulty nodes. Then $r$ can broadcast a message to every non-faulty node $v$ in $M$ by broadcasting the message over all the $k$ trees. Since the number of faulty nodes is less than $k$, at least one of the $k$ internally disjoint paths from $r$ to $v$ is fault free; this assures the message delivery to every node in the network;
- In secure message distribution, a message can be divided into $k$ packets where each packet is sent by node $r$ to its destination using a different spanning tree. Thus, each node in the network receives at most one of the $k$ packets except the destination node that receives all the $k$ packets.

In this regard, the problem of constructing ISTs on graphs is worth studying. Zehavi and Itai proposed that $k$ ISTs can be constructed from a $k$-connected graph [19]. The conjecture has been proved true for $k$-connected graphs with $k \leq 4$ [20]–[22], but it remains open for $k \geq 5$. However, the problem is very tough for arbitrary graphs, and researchers have shifted their attention to methods for constructing ISTs in interconnection networks over the past decade.

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.

**TABLE 1. Comparison of methods in different interconnection networks whose dimension is *n* and the number of nodes is *N*.**

| Interconnection networks | The determination of one node's parent | Time Complexity |
|---|---|---|
| Twisted cubes [23] | by rules | $O(nN)$ |
| Cross cubes [24] | by rules | $O(n^2N)$ |
| Möbius cubes [25] | by rules | $O(nN)$ |
| Locally twisted cubes [4] | by rules | $O(nN)$ |
| Parity cubes [26] | by rules | $O(nN)$ |
| Hypercubes [7] | by rules | $O(nN)$ |
| Folded hypercubes [9] | by rules | $O(nN)$ |
| Bubble-sort networks [10] | by rules | $O(n^2N)$ |
| Alternating group networks | no need | $O(n^2N)$ |

A class of graphs called group graphs or Cayley graphs are crucial for the design and analysis of interconnection networks for parallel and distributed computing. [12]–[15]. The IST problem has been solved on several interconnection networks, including twisted cubes [23], cross cubes [24], Möbius cubes [25], locally twisted cubes [4], [5], parity cubes [26], hypercubes [7], [8], folded hypercubes [9], and bubble-sort networks [10], [11]. To the best of our knowledge, the approach of constructing ISTs in alternating group networks is still unknown. In this paper, we focus on alternating group networks.

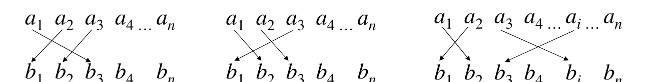
The alternating group networks $AN_n$ ($n$ stands for dimensions) proposed by Youhu in 1998 [17] differ from the alternating group graphs $AG_n$ introduced by Jwo *et al.* in 1993 [16]. The new alternating group networks are also Cayley graphs and are thus vertex-symmetric. The diameters of $AN_n$ and $AG_n$ are comparable; however, the node degree of $AN_n$ is only about half that of $AG_n$. Furthermore, the new graphs are maximally fault-tolerant and share some of the positive structural attributes of the well-known star graphs [15]. $AN_n$ is a regular graph with $n!/2$ nodes and $n!(n-1)/4$ edges as well as a node degree $n-1$. $AN_n$ is Hamiltonian and has a diameter for which the upper bound is $3(n-2)/2$ [18].

In this paper, we propose a novel and simple top-down approach for constructing ISTs in alternating group networks. The main ideas of the algorithm are to use induction to develop small trees to big trees, to use a triangle breadth-first search (TBFS) process to create a backbone of an IST, and to use breadth-first search (BFS) process to connect the rest of nodes. Compared to other methods of different interconnection networks in the literature presented in Table 1, the uniqueness of our method is that it does not need to determine the parent of one node by any rule, on the contrary others determine that by rules. The time complexity in $n$-dimensional alternating group network $AN_n$ is O($n^2 \times n!$), where $n!$ is twice the number of nodes of $AN_n$; hence it is polynomial time.

The remainder of this paper is organized as follows: Section II explains preliminary considerations; Section III presents the algorithm; Section IV proves the relevant claims; Section V describes the software implementation; and Section VI concludes the paper.

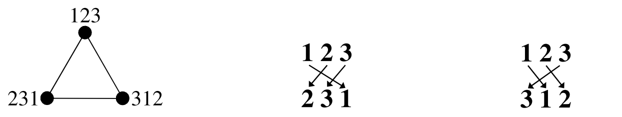
## II. PRELIMINARY CONSIDERATIONS

An $n$-dimensional alternating group network $AN_n$ is defined to be a Cayley graph $G = G(V, E)$ in an alternating group $A_n$,


$a_1$ $a_2$ $a_3$ $a_4$ … $a_n$
$b_1$ $b_2$ $b_3$ $b_4$ … $b_n$
$a_1$ $a_2$ $a_3$ $a_4$ … $a_n$
$b_1$ $b_2$ $b_3$ $b_4$ … $b_n$
$a_1$ $a_2$ $a_3$ $a_4$ … $a_i$ … $a_n$
$b_1$ $b_2$ $b_3$ $b_4$ … $b_i$ … $b_n$

(a) Left child of node $a$. (b) Right child of node $a$. (c) Cross node $i$ of node $a$.

**FIGURE 1. Three operations in alternating group networks.**



(a) $AN_3$. (b) Left child. (c) Right child.

**FIGURE 2. $AN_3$ and children of node 123.**

where $V$ is the set of all even permutations of $\langle n \rangle = \{1, 2, \ldots, n\}$ and $E$ consists of symmetric edges $(u, v)$ such that any two distinct permutations $u$ and $v$ are connected by an edge if and only if one can be reached from the other through operations such as left child, right child, and cross node. Suppose that $n \geqq 3$, $AN_n$ has the vertex set of even permutations from $\{1, 2, \ldots, n\}$; two vertices $[a_1, a_2, a_3, \ldots, a_n]$ and $[b_1, b_2, b_3, \ldots, b_n]$ are adjacent if one of the following three conditions is satisfied.

- The first condition is $a_1 = b_3, a_2 = b_1, a_3 = b_2$ and $a_j = b_j$ for $4 \leq j \leq n$. As illustrated in Fig. 1(a), node $b$ is the left child of node $a$.
- The second condition is $a_1 = b_2, a_2 = b_3, a_3 = b_1$ and $a_j = b_j$ for $4 \leq j \leq n$. As presented in Fig. 1(b), node $b$ is the right child of node $a$.
- The third condition is that there exists $i \in \{4, 5, \ldots, n\}$ such that $a_1 = b_2, a_2 = b_1, a_3 = b_i, a_i = b_3$, and $a_j = b_j$ for $j \in \{4, 5, ..., n\} \backslash \{i\}$. As shown in Figure 1(c), node $b$ is cross node $i$ of node $a$.

$AN_3$ is shown in Figure 2(a). Because odd permutations are prohibited and because the permutation 213, 132, 321 are all odd permutations, node 213, node 132, and node 321 do not exist in $AN_3$. Accordingly, $AN_3$ has three nodes and there edges and is a triangle. Node 123 has two children, as illustrated in Figure 2(b) and 2(c).

$AN_4$ is displayed in Figure 3. The set {1, 2, 3, 4} has 24 permutations, namely 12 even permutations and 12 odd permutations. We require only even permutations; hence, $AN_4$ has 12 nodes and 18 edges. Node 1234 has two children and one cross node as presented in Figure 4. $AN_5$ consists of 60 nodes and 120 edges, as displayed in Figure 5, and is composed of five instances $AN_4$.

## III. ALGORITHM

Notations used in the following paragraphs:

- the root: node 123456 … $n$;
- group $A$: a set of nodes whose last character is $A$;
- the root group: the set of nodes whose last character is identical to that of the root. In $AN_n$, the root group is group $n$;
- $T_j^n$: the $j$th IST of $AN_n$, and its backbone whose last character is $j$;
- *tid*: the last character of the bone seed, namely $j$ of $T_j^n$.

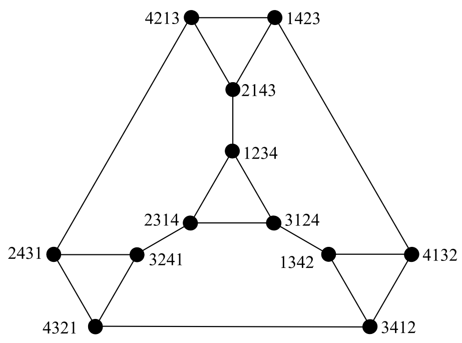
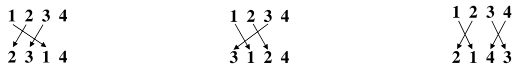


**FIGURE 3.** ***AN*₄.**



(a) Left child. (b) Right child. (c) Cross node 4.

**FIGURE 4. Children and cross node of node 1234 in $AN_4$.**

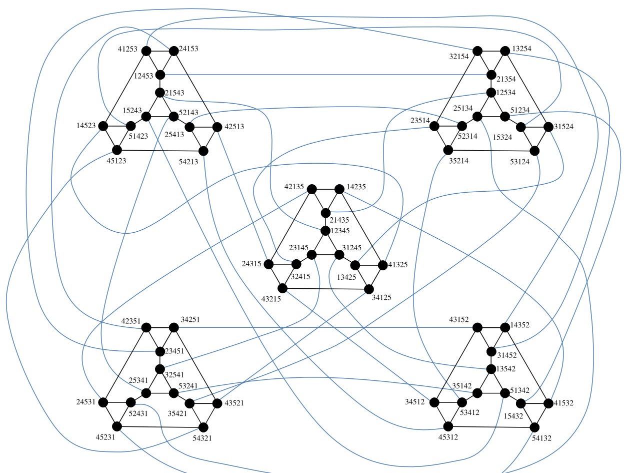


**FIGURE 5.** **$AN_5$.**

$AN_n$ is partitioned into $n$ instances of $AN_{n-1}$. Every node is classified into a group by its last character. Thus, $AN_n$ has $n$ groups. For example, $AN_4$ can be divided into four instances of $AN_3$. $AN_4$ has four groups, as illustrated in Figure 6.

Each node in $AN_{n-1}$ can be transformed into some node in $AN_n$ if one character '$n$' is appended to the tail of the sequence in the $n$ position. We thus construct the ISTs of $AN_n$ by taking advantage of that of $AN_{n-1}$.

In $AN_n$, the root has $n-1$ edges and connects $n-1$ nodes. In the $n-1$ nodes connected by the root, $n-2$ nodes can be constructed from those in $AN_{n-1}$ if one character '$n$' is appended to the $n$ position, but the cross node $n$ is a new node. Each of the $n-2$ nodes has a new edge that connects to a node that is called a **bone seed** and the cross node $n$ itself is also a bone seed. We use bone seeds to create a backbone to connect other groups.

The concept is illustrated in Figure 7. Each of the central white nodes can be formed from some node in $AN_{n-1}$ if a single character '$n$' is appended to the $n$ position. The left

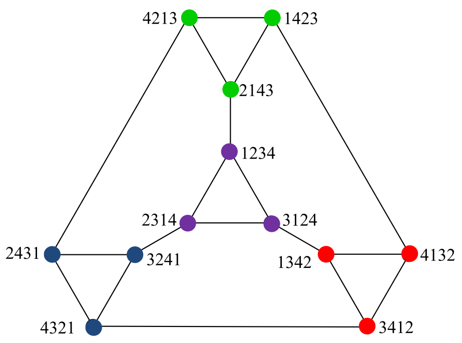


**FIGURE 6. Colored depiction of $AN_4$.**

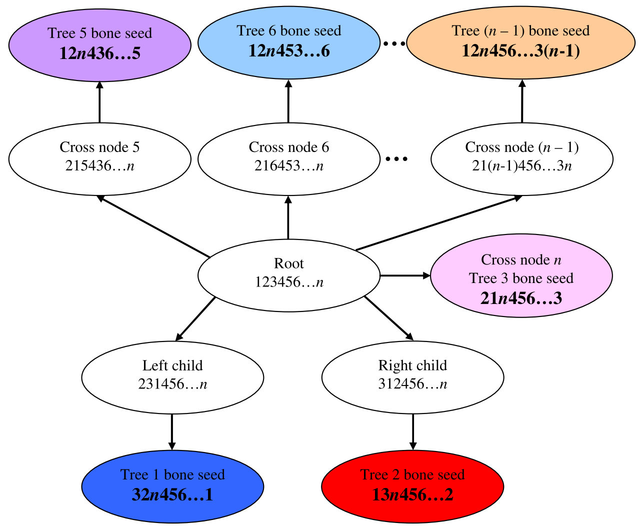


**FIGURE 7. Bone seeds.**

child of the root in $AN_n$ uses its new edge to make the bone seed of $T_1^n$. The right child of the root in $AN_n$ uses its new edges to make the bone seed of $T_2^n$. The root in $AN_n$ uses its new edge to make the bone seed of $T_3^n$. The cross nodes 5 to $n-1$ of the root in $AN_n$ use their new edges to make the bone seeds of $T_5^n$ to $T_{n-1}^n$. $T_4^n$ does not conduct any bone function; consequently, it has no bone seed.

## A. MAIN ALGORITHM

We use the following global variables in Algorithm 1.

- *tree*: an array of the trees of this iteration; it is a two-dimensional array, and *tree*[ *id* ][ *node* ] represents its parent. Therefore, every tree includes all nodes. If the node has not yet been visited, its value is 0; *tree*[ *id* ] [ *node* ] = 0.
- *ptree*: an array of the trees of the previous iteration.
- *arrow*: an associative array for storing all direct edges. Initially, an *arrow*[ *from* ][ *to* ] value of 1 represents an unused direct edge; an *arrow*[ *from* ][ *to* ] value of 0 represents a used direct edge.

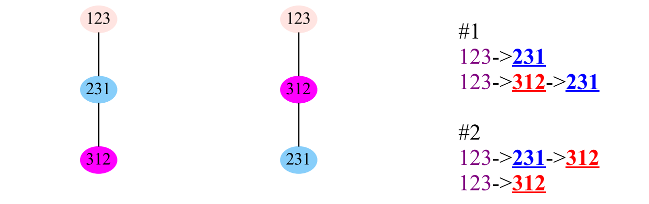
$AN_3$**, the basic part,** is a triangle, having two independent spanning trees that are illustrated in Figure 8.

**Algorithm 1** Main Algorithm

```
Input  : size: problem size, ex. 10 means $AN_3$ to $AN_{10}$
Output: the ISTs from $AN_3$ to $AN_{size}$
for $t = 3; t \leq size; t = t + 1$ do
    if $t == 3$ then
        basic part, a triangle;
    else if $t \geq 4$ then
        Copy previous trees of the previous iteration to
        the trees of this iteration;
        Make a bonelist;      ▷ an array to store the bone
                                                  ▷ seeds
        for every bone seed in the bonelist do
            Execute bone operation;
        //spark block
        if $t == 4$ then
            for $i = 0; i < 2; i = i + 1$ do
                for from tree 1 to tree 3 do
                    if $i == 0$ then
                        spark(tree id, true);
                    else
                        spark(tree id, false);
        else if $t \geq 5$ then
            spark(4, true);                       ▷ (1)
            // Spark three times.                 ▷ (2)
            for $i = 0; i < 3; i = i + 1$ do
                for $j = 1; j \leq t - 1; j = j + 1$ do
                    if $j \neq 4$ then
                        if $i == 0$ then
                            spark($j$, true);
                        else
                            spark($j$, false);
                spark(4, false);
            spark(3, false);                      ▷ (3)
    ptree = tree;
/*
Comments:
(1) First, Tree 4 executes a spark operation to produce
cross nodes along the root group, Tree 4 first occurs in
$AN_5$, and it copies Tree 3 of $AN_4$ to its own tree array,
namely tree[4].
(2) Second, every tree executes spark operations, and
Tree 4 is the last tree to execute those operations.
(3) Relative to other trees, Tree 3 requires one more
spark operation to connect the root group.
*/
```

### 1) COPY PREVIOUS TREES

We use an array to store trees in every iteration. We implement the algorithm in PHP and employ PHP associative arrays. Associative arrays differ from numeric arrays in that associative arrays use descriptive names for id keys.

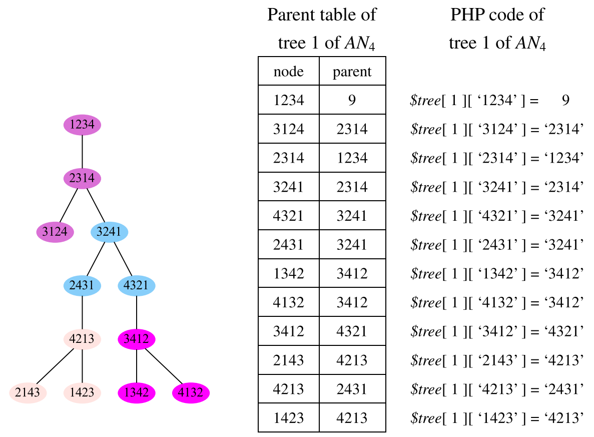


#1
123->231
123->312->231

#2
123->231->312
123->312

(a) Tree 1 of $AN_3$. (b) Tree 2 of $AN_3$. (c) All paths in both trees.

**FIGURE 8.** ISTs in $AN_3$ and all paths.



Parent table of tree 1 of $AN_4$

| node | parent |
|---|---|
| 1234 | 9 |
| 3124 | 2314 |
| 2314 | 1234 |
| 3241 | 2314 |
| 4321 | 3241 |
| 2431 | 3241 |
| 1342 | 3412 |
| 4132 | 3412 |
| 3412 | 4321 |
| 2143 | 4213 |
| 4213 | 2431 |
| 1423 | 4213 |

PHP code of tree 1 of $AN_4$

```
$tree[ 1 ][ ‘1234’ ] =    9
$tree[ 1 ][ ‘3124’ ] = ‘2314’
$tree[ 1 ][ ‘2314’ ] = ‘1234’
$tree[ 1 ][ ‘3241’ ] = ‘2314’
$tree[ 1 ][ ‘4321’ ] = ‘3241’
$tree[ 1 ][ ‘2431’ ] = ‘3241’
$tree[ 1 ][ ‘1342’ ] = ‘3412’
$tree[ 1 ][ ‘4132’ ] = ‘3412’
$tree[ 1 ][ ‘3412’ ] = ‘4321’
$tree[ 1 ][ ‘2143’ ] = ‘4213’
$tree[ 1 ][ ‘4213’ ] = ‘2431’
$tree[ 1 ][ ‘1423’ ] = ‘4213’
```

(a) Tree 1 of $AN_4$. (b) Parent table of tree 1 of $AN_4$ and PHP code.

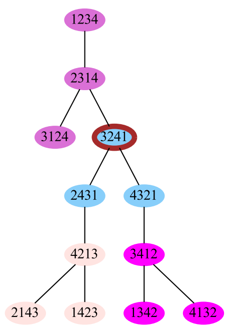
**FIGURE 9.** Tree 1 of $AN_4$, its parent table, and PHP code.

**TABLE 2.** Tree id and previous tree id mapping.

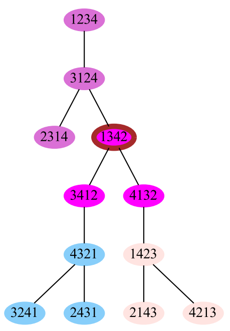
| Tree id | 1 | 2 | 3 | 4 | 5 | 6 | … | $n-1$ |
|---|---|---|---|---|---|---|---|---|
| Previous tree id | 1 | 2 | no previous tid, a new tree | 4 | 5 | 6 | … | 3 |

Array keys serve to identify nodes with values storing their parents. For example, in $AN_4$, the parent of node 3241 is node 2314 in Tree 1, the information is stored in an array, namely *tree*[ 1 ][ ’3241’ ] = ’2314’. These quotation marks are used to indicate string constants. At the end of each iteration, the corresponding trees must be stored in another array *ptree* so that correct information can be used in the next iteration. The tree array *ptree* stores the previous trees. At the start of every iteration, the program should copy previous trees to the tree array *tree*. The data structure of each tree uses an array to associate nodes with their parents. Tree 1 of $AN_4$ is shown in Figure 9. Note that the parent of node 1234 is set to 9, a dummy value to prevent the root from being visited again.

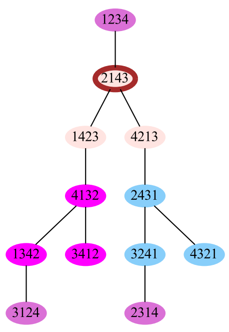
Every tree has a tree id. Each tree id is the last character of the bone seed. Table 2 presents the mapping a new tree and an old tree. In a typical iteration, $T_1^n$ copies information from $T_1^{n-1}$ and $T_{n-1}^n$ copies information from $T_3^{n-1}$. $T_3^n$ is a new tree, and thus no previous tree is copied.



(a) Tree 1 bone seed: 3241.



(b) Tree 2 bone seed: 1342.



(c) Tree 3 bone seed: 2143.

**FIGURE 10.** Bone seeds (nodes with dark brown outlines) in $AN_4$.

## 2) CREATE A BONELIST

A *bonelist* is an array for storing the bone seeds. The bone seeds are created from the root (123456...). In $AN_4$, we can create three bone seeds, namely 3241, 1342, 2143 to establish backbones, as shown in Figure 10.

For example, $AN_5$ has four trees and three bone seeds: 32541 in $T_1^5$, 13542 in $T_2^5$, 21543 in $T_3^5$, as presented Figures 13, 14, and 15, respectively. As indicated in Figure 16, Tree 4 does not execute any bone operation, but it does conduct spark operations.

## B. BONE FUNCTION

A backbone tree can be constructed from a bone seed through a triangle breadth-first search (TBFS) process. However, the TBFS process must stop when it encounters a node of which the last character differs from that of the bone seed. First, we review the BFS process. The BFS is a graph traversal approach using a queue, in contrast to a depth-first search process, which uses a stack. For example, Figure 11 presents a graph that can be traversed using the BFS process.

**BFS order.** In the bone function, every node in the bone queue must visit all hitherto unvisited nodes connected to it in the following order:

1. left child; 2. right child; 3. cross nodes 4, 5, 6, ..., $n$.

**TBFS process.** Because $AN_n$ is symmetric, the graph illustrated in Figure 12(a) is derived. We remove the last visited

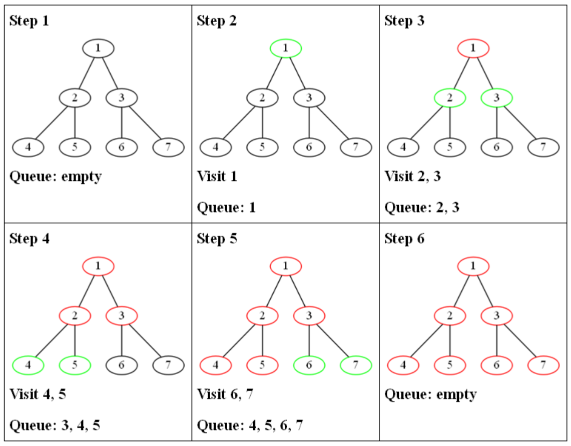
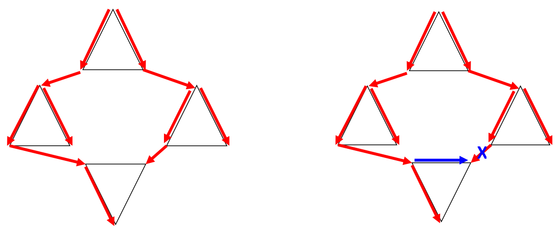


**FIGURE 11.** BFS steps.



(a) BFS (b) TBFS

**FIGURE 12.** BFS and TBFS processes.

edge (blue X) and use another edge (blue arrow) to visit the node. Thus, the triangle shape is preserved. Figure 12(b) presents this concept.

The Basic functions corresponding to the aforementioned processes are presented in the following passages. The bone procedure is specified in Algorithm 2.

- function leftchild(*parent*): returns the left child of *parent*;
- function rightchild(*parent*): returns the right child of *parent*;
- function crossnode(*parent*, $p$): returns the cross node $p$ of *parent* by swapping first and second characters, as well as the third and the $p$th ($p \neq 1, 2, 3$) characters of *parent*.

For example, in Figure 17, the backbone nodes and boundary nodes of $T_1^5$ are indicated by ellipses with blue and brown borders respectively. When the TBFS process encounters different color nodes whose last character is different from the bone seed, it does not put them into the queue. These nodes are called boundary nodes.

## C. SPARK FUNCTION

The spark functions shown in Algorithm 3 must connect all unvisited nodes by one step at a time from a tree after the bone operation. For example, in Figure 18, the spark nodes of $T_1^5$ in the first and second spark executions are indicated by ellipses with gold and gray borders respectively.

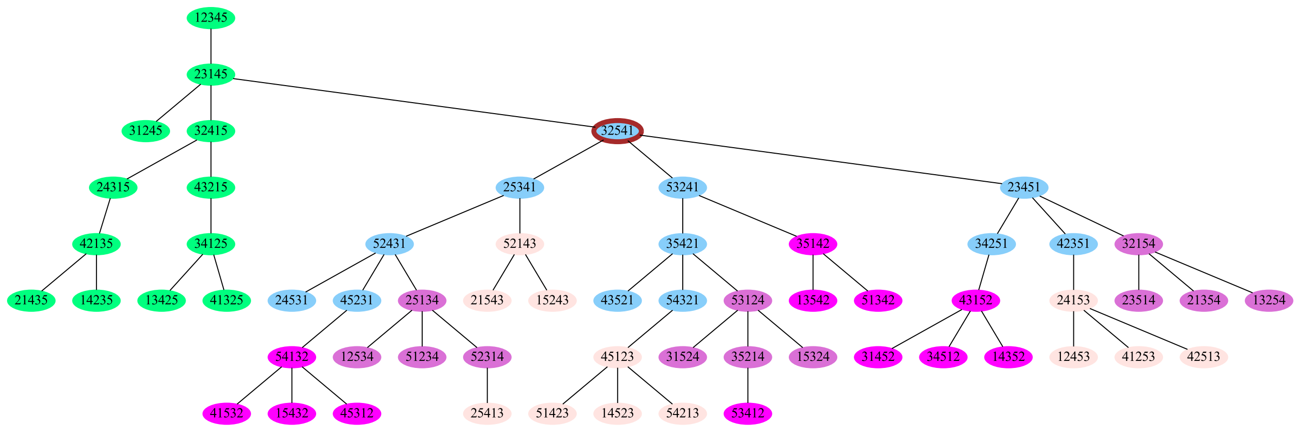


**FIGURE 13.** **Tree 1 bone seed: 32541 in $AN_5$.**

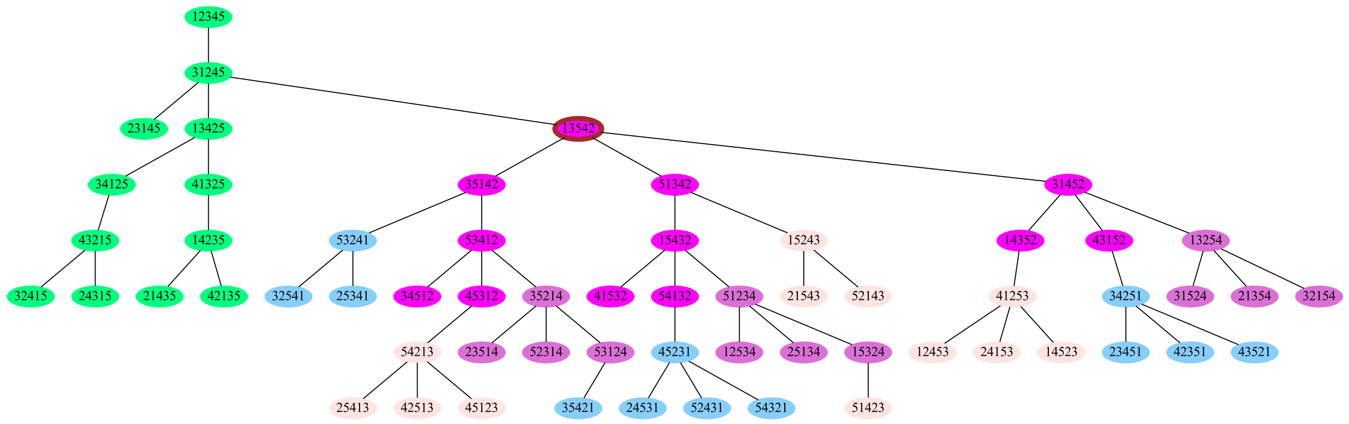


**FIGURE 14.** **Tree 2 bone seed: 13542 in $AN_5$.**

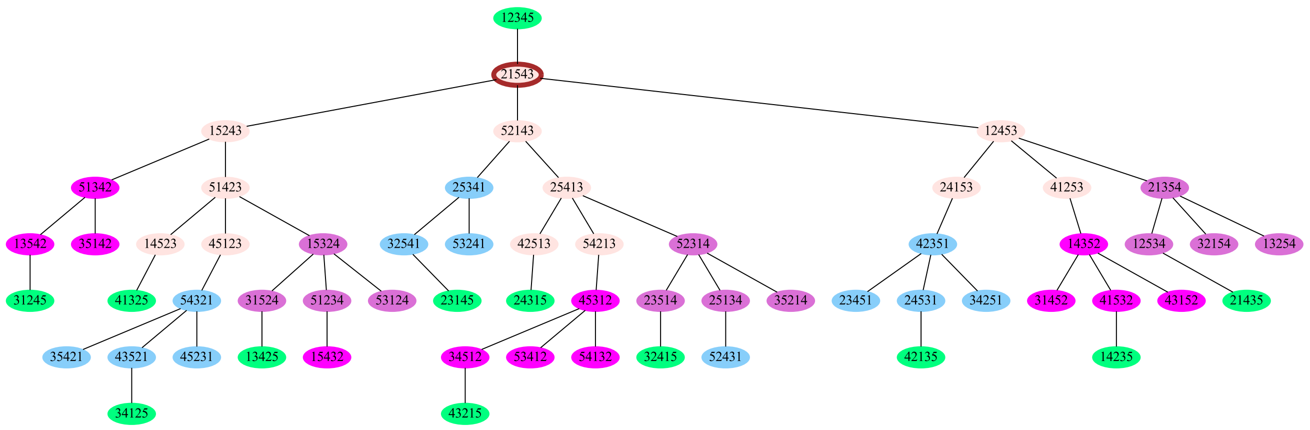


**FIGURE 15.** **Tree 3 bone seed: 21543 in $AN_5$.**

## *D. PATH COMPOSITION*

According to the algorithm, the output paths from the root to other nodes include the backbone part and the spark part, as displayed in Figure 19. The backbones of trees except for Tree 4 have a bone seed.

Because Tree 4 done not execute a bone operation, the last character of the root is appended to its previous tree as its backbone. For example, in Tree 4 in $AN_5$ (Figure 16), the green nodes originate from Tree 3 in $AN_4$, and the last character ’5’ is appended to construct the backbone in $AN_5$.

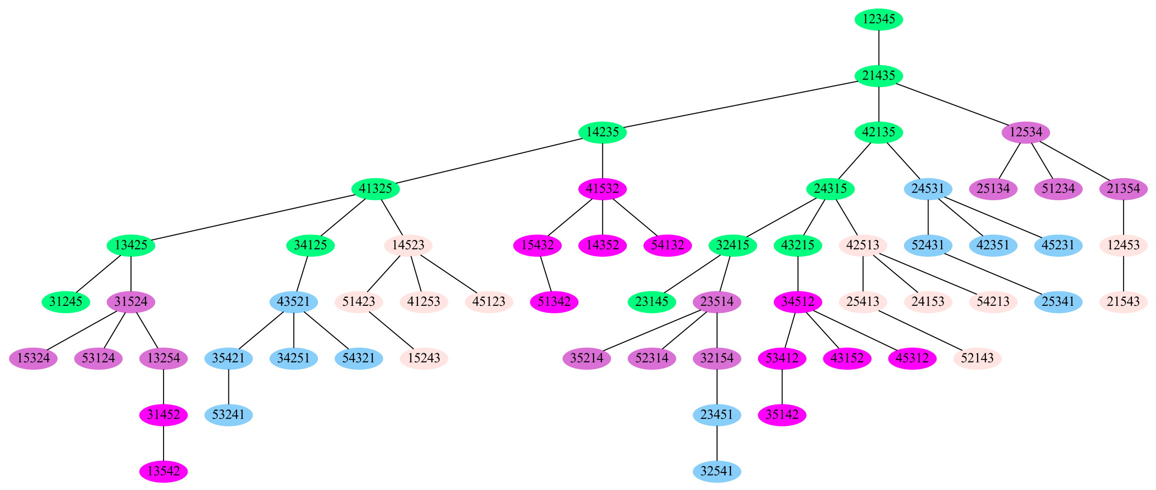


**FIGURE 16.** Tree 4 in $AN_5$ has no bone seed due to no bone operation.

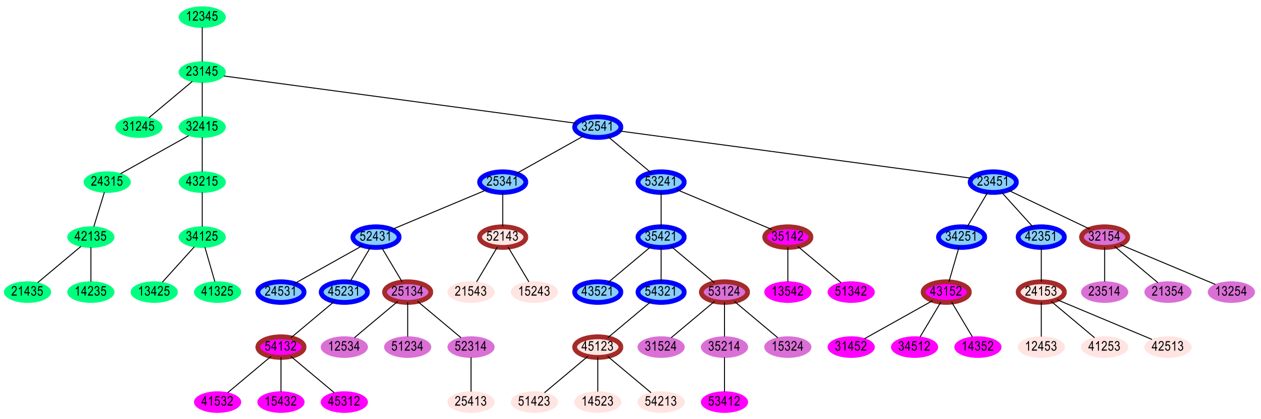


**FIGURE 17.** Backbone nodes (blue border) and boundary nodes (brown border) of $T_1^5$ after bone function.

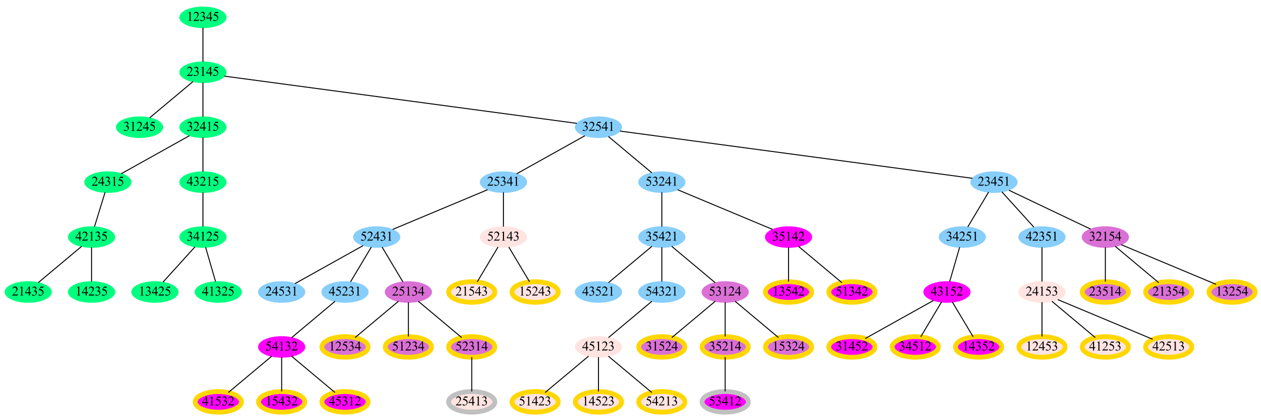


**FIGURE 18.** Spark nodes (gold and gray border) of $T_1^5$ after spark function.

## IV. PROOF

*Lemma 1:* Every node in $AN_n$ belongs to a triangle.

*Proof:* Assume that one node *abcd* belongs to two triangles. In Figure 20, nodes *abcd*, *bcad*, and *cabd* form a triangle with characters *a*, *b*, and *c*. Node *abcd* connects to the cross node *badc*. The triangle with characters *b*, *a*, and *d* cannot have another edge connected to node *abcd*. Accordingly, the aforementioned assumption is impractical. ■

**Algorithm 2** Create a Backbone

```
Input: seed and tid                          ▷ the bone seed and tree id
Function bone (seed, tid)
    boneQueue = array(seed);
                          ▷ put seed into boneQueue initially
    /*
    From node seed, search left child, right child and
    cross nodes. During searching,if the visited node is
    in the same group as seed, it will be put into
    boneQueue. */
    for $j = 0; j < |boneQueue|; j = j + 1$ do
        v = boneQueue[j];             ▷ get the jth element of
                                                  ▷ boneQueue
        lchild = leftchild(v);
        rchild = rightchild(v);
        // Case 1: visit left and right children
        // left and right children are unvisited;
        // both edges from v to them are unused
        if tree[tid][lchild] == 0 and
        arrow[v][lchild] == 1 and
        tree[tid][rchild] == 0 and
        arrow[v][rchild] == 1 then
            // visit left child
            tree[tid][lchild] = v;                ▷ (1)
            arrow[v][lchild] = 0;                 ▷ (2)
            boneQueue[] = lchild;                 ▷ (3)
            // visit right child
            tree[tid][rchild] = v;                ▷ (1)
            arrow[v][rchild] = 0;                 ▷ (2)
            boneQueue[] = rchild;                 ▷ (3)
        end
        // Case 2: TBFS
        // left child is unvisited, its edge unused;
        // right child is visited, its edge unused
        if tree[tid][lchild] == 0 and
        arrow[v][lchild] == 1 and
        tree[tid][rchild] ≠ 0 and
        arrow[v][rchild] == 1 then
            // visit left child
            tree[tid][lchild] = v;                ▷ (1)
            arrow[v][lchild] = 0;                 ▷ (2)
            boneQueue[] = lchild;                 ▷ (3)
            //
            // set the edge (from the right child's
            // parent to the right child) unused
            // (from 0 to 1)
            arrow[tree[tid][rchild]][rchild]++;
            tree[tid][rchild] = 0;                ▷ (4)
            // visit right child
            tree[tid][rchild] = v;                ▷ (1)
            arrow[v][rchild] = 0;                 ▷ (2)
            boneQueue[] = rchild;                 ▷ (3)
        end
        // continued on next page

        //Case3: TBFS
        //left child is visited, its edge unused;
        //right child is unvisited, its edge unused
        if tree[tid][lchild] ≠ 0 and
        arrow[v][lchild] == 1 and
        tree[tid][rchild] == 0 and
        arrow[v][rchild] == 1 then
            // set the edge (from the left child's
            // parent to the left child) unused
            // (from 0 to 1)
            arrow[tree[tid][lchild]][lchild]++;
            tree[tid][lchild] = 0;                ▷ (4)
            // visit the left child
            tree[tid][lchild] = v;                ▷ (1)
            arrow[v][lchild] = 0;                 ▷ (2)
            boneQueue[] = lchild;                 ▷ (3)
            // visit the right child
            tree[tid][rchild] = v;                ▷ (1)
            arrow[v][rchild] = 0;                 ▷ (2)
            boneQueue[] = rchild;                 ▷ (3)
        end
        // visit cross nodes
        for $i = 4$; $i \leq$ the number of characters of one
        node; $i = i + 1$ do
            cnode = crossnode( v, i );
            if tree[tid][cnode] == 0 and
            arrow[v][cnode] == 1 then
                tree[tid][cnode] = v;             ▷ (1)
                arrow[v][cnode] = 0;              ▷ (2)
                if the last character of v == the last
                character of cnode then
                    boneQueue[] = cnode;          ▷ (3)
                end
            end
        end
    end
    /* Comments:
    (1) Set the parent of the left child, right child,
    or cross nodes.
    (2) Set the edge used (from 1 to 0).
    (3) Put the left child, right child or cross nodes into
    boneQueue.
    (4) Set the right or left child unvisited. */
End Function
```

*Lemma 2:* If one node has one child (either right or left) in the backbone part, it must have the other child.

*Proof:* Because we use the TBFS process in the bone function, every parent in the backbone will have two children. ■

**Algorithm 3** Connect the Unvisited Nodes by One Step at a Time From Tree *tid*

```
Input: $tid$ and $first$                        ▷ tree id and
                                  ▷ first execution (boolean value)
Function spark ($tid, first$)
    // global means using global variable
    global $spkQue$;                ▷ a two-dimensional array for
        ▷ storing nodes visited by Tree $tid$ not to traverse
                            ▷ all nodes in Tree $tid$ each time
    global $spkQueIndex$;                 ▷ the start position in
                          ▷ $spkQue[tid]$ of Tree $tid$ each time
    if $first == true$ then
        // first execution
        for every node $v$ in $tree[tid]$ do
            //only Tree 4 can grow from the root group
            if $v$ has parent and ( ( $tid \neq 4$ and the last
            character of $v \neq$ the last character of the
            root) or $tid == 4$) then
                $lch$ = leftchild( $v$ );
                $rch$ = rightchild( $v$ );
                // visit the left child
                if $tree[tid][lch] == 0$ and
                $arrow[v][lch] == 1$ then
                    $tree[tid][lch] = v$;               ▷ (1)
                    $arrow[v][lch] = 0$;                ▷ (2)
                    $spkQue[tid][] = lch$;              ▷ (3)
                end
                // visit the right child
                if $tree[tid][rch] == 0$ and
                $arrow[v][rch] == 1$ then
                    $tree[tid][rch] = v$;               ▷ (1)
                    $arrow[v][rch] = 0$;                ▷ (2)
                    $spkQue[tid][] = rch$;              ▷ (3)
                end
                // visit the cross nodes
                for $i = 4$; $i \leq$ the number of characters
                of one node; $i = i + 1$ do
                    $cn$ = crossnode( $v, i$);
                    if $tree[tid][cn] == 0$ and
                    $arrow[v][cn] == 1$ then
                        $tree[tid][cn] = v$;            ▷ (1)
                        $arrow[v][cn] = 0$;             ▷ (2)
                        $spkQue[tid][] = cn$;           ▷ (3)
                    end
                end
            end
        end

    else
        $qsize$ = count( $spkQue[tid]$ );
                              ▷ current size of $spkQue[tid]$
        /*
        new element will be put into $spkQue[tid]$, but
        will be checked in next execution
        */
        for $y = spkQueIndex[tid]$; $y < qsize$;
        $y = y + 1$ do
            $v = spkQue[tid][y]$;
            if ($tid \neq 4$ and the last character of $v \neq$ the
            last character of the root ) or $tid == 4$ then
                $lch$ = leftchild( $v$ );
                $rch$ = rightchild( $v$ );
                // visit the left child
                if $tree[tid][lch] == 0$ and
                $arrow[v][lch] == 1$ then
                    $tree[tid][lch] = v$;               ▷ (1)
                    $arrow[v][lch] = 0$;                ▷ (2)
                    $spkQue[tid][] = lch$;              ▷ (3)
                end
                // visit the right child
                if $tree[tid][rch] == 0$ and
                $arrow[v][rch] == 1$ then
                    $tree[tid][rch] = v$;               ▷ (1)
                    $arrow[v][rch] = 0$;                ▷ (2)
                    $spkQue[tid][] = rch$;              ▷ (3)
                end
                // visit the cross nodes
                for $i = 4$; $i \leq$ the number of characters
                of one node; $i = i + 1$ do
                    $cn$ = crossnode( $v, i$ );
                    if $tree[tid][cn] == 0$ and
                    $arrow[v][cn] == 1$ then
                        $tree[tid][cn] = v$;            ▷ (1)
                        $arrow[v][cn] = 0$;             ▷ (2)
                        $spkQue[tid][] = cn$;           ▷ (3)
                    end
                end
            end
        end
        $spkQueIndex[tid] = y$;
                       ▷ store the position for next execution
    end
    /* Comments:
    (1) Set the parent of the left child, right child,
    or cross nodes.
    (2) Set the edge used (from 1 to 0).
    (3) Put the left child, right child or cross nodes into
    $spkQue[tid]$. */
End Function
```

*Lemma 3:* Every node in $AN_n$ provides an incoming edge to every tree.

*Proof:* $AN_n$ is connected and has $n - 1$ trees. If some node provides its two incoming edges to some tree, then the tree will visit this node twice. The concept is illustrated in Figure 21, where different colors mean different trees. ■

*Lemma 4:* Every tree can visit all nodes in group 4 in most two steps.

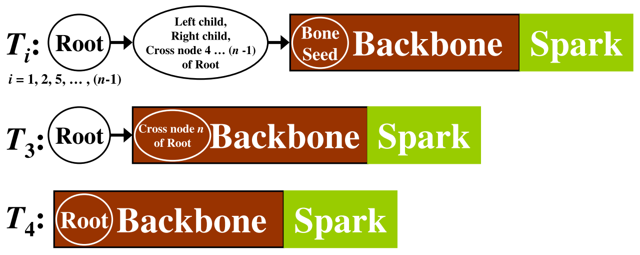


**FIGURE 19.** **Path composition.**

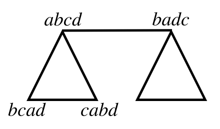


**FIGURE 20.** **Each node belongs to one triangle.**

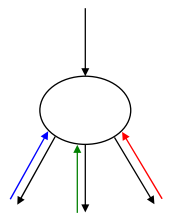


**FIGURE 21.** **Incoming and outgoing edges.**

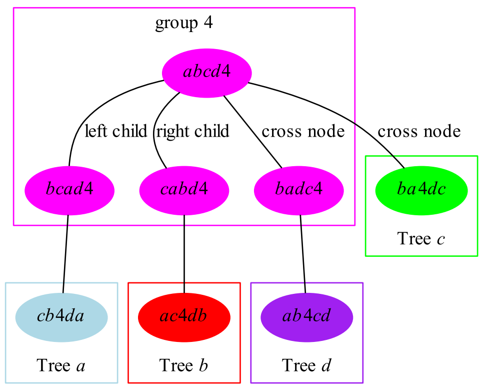


**FIGURE 22.** **Connections of group 4.**

*Proof:* Tree 4 does not conduct any bone operation. The edges in group 4 are all unused. Every node in group 4 is connected directly by some tree. If some node in group 4 is not directly connected by some tree, it can connect its left child, right child and cross nodes to reach some tree. Consider one node *abcd*4 in group 4. This node will be connected directly by tree *c*, and trees *a*, *b*, and *d* reach it in two steps, as displayed in Figure 22. ∎

*Lemma 5:* The spark part has five patterns: P1, P2, P3, P4, and P5, as displayed in Figure 23.

*Proof:* **P1:**

The node in P1 is a boundary node, the third character of which is the same as the last character of the objective backbone nodes.

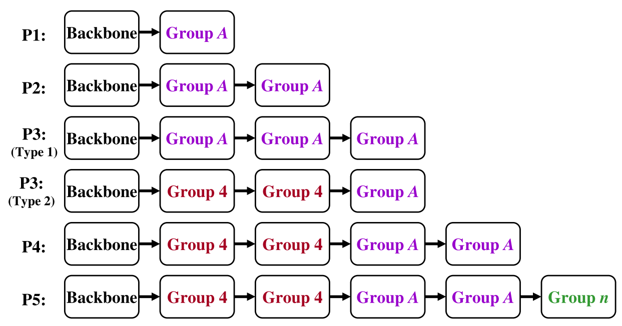


**FIGURE 23.** **Five spark patterns.**

**P2:**

The node in P2 can reach tree backbones by its left child, right child, or cross nodes according to the appropriate tree id.

**P3:**

Assume that node *v* whose last character is *s* must have an edge *e* to walk to the backbone of Tree *t* through the character *t*. However, *e* has been used in the backbone of Tree *s*. Because Tree 4 does not execute any bone operations and it uses the last character *n* of the root as the last character of its backbone nodes, node *v* will utilize the character 4 to walk an edge to the next node *u* according to Lemma 4. Subsequently, node *u* can utilize its character *t* to the backbone of Tree *t*. The following examples illustrate P1, P2, and P3(type 1 and type 2).

Ex. $AN_6$: five paths from the root to node 625431 can be calculated as follows:

The last character of node 625431 is 1; therefore, this node is the backbone node of Tree 1. The parent of node 625431 is node 263451 in Tree 1; the parent node uses 3 to traverse node 625431, and consequently Tree 3 cannot use 3 to traverse node 625431. Node 625431 uses 4 to approach the backbone of Tree 3.

123456→231456→326451→263451→625431 (backbone)
123456→312456→136452→613452→165432→651432 →562431→625431 (P2)
123456→216453→621453→264153→625143→261543 →623541→264531→625431(P3 type1)
123456→214356→421356→243156→425136→254136 →521436→256431→625431(P2)
123456→215436→126435→261435→625431 (P1)

Ex. $AN_6$: five paths from the root to node 624351

The last character of node 624351 is 1; therefore, this node is the backbone node of Tree 1. The parent of node 624351 is node 263451 in Tree 1; this parent uses 3 to traverse node 624351, and consequently, Tree 3 cannot use 3 to traverse node 624351. Node 624351 uses 4 to approach the backbone of Tree 3.

123456→231456→326451→263451→624351 (backbone)
123456→312456→136452→613452→164352→641352 →462351→624351 (P2)
123456→216453→621453→264153→623154→261354 →624351 (P3 type2)

123456→214356→421356→246351→624351 (P2)
123456→215436→126435→261435→624135→263145
→621345→265341→624351(P2)

**P4:**

Similar the case of P3, node $v$ whose last character is $s$ must have an edge $e_1$ to walk to the backbone of Tree $t$ through the character $t$. However, $e_1$ is used in the backbone of Tree $s$. Node $v$ will utilize the character 4 to walk an edge to the next node $u$ according to Lemma 3. Node $u$ subsequently requires an edge $e_2$ to walk to the backbone of Tree $t$ through the character $t$. However, the edge $e_2$ is still used in the backbone of Tree $s$. $u$ will utilize the character 4 to walk an edge to the next node $w$ according to Lemma 3. Here, the third character of node $u$ is 4; therefore the last character of node $w$ is 4. Node $w$ will utilize its character $t$ to reach the backbone of Tree $t$. According to Lemma 4, node $w$ will just pass one node in group 4 to reach the backbone of Tree $t$. P4 is demonstrated in the following example.

123456→231456→326451→632451→36$\underline{5}$421→653421 (backbone)
123456→312456→136452→613452→165432→651432
→562431→653421 (P2)
123456→216453→162453→615423→561423→653421 (P1)
123456→214356→421356→243156→432156→345126
→534126→351426→536421→653421(P2)
123456→215436→126435→612435→163425→614325
→165324→651324→564321→653$\underline{4}$21(P4)

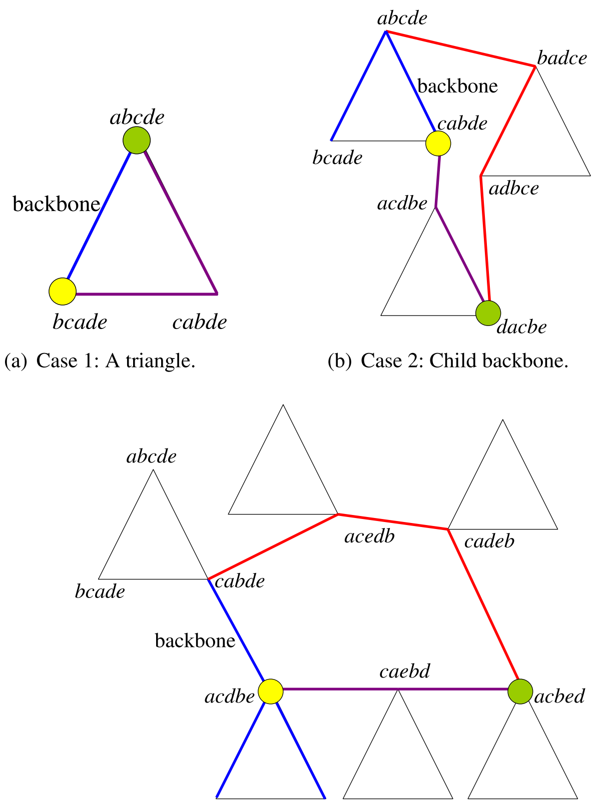
We assess the parent of node 564321 in $T_1^6$. We determine that its parent is node 645321 and that the third character of node 645321 is still 5. Therefore, node 564321 in $T_5^6$ should utilize the character 4 to approach group 5.

123456→231456→326451→263451→624351→462351
→64$\underline{5}$321→564321 (backbone)
123456→312456→136452→613452→164352→615342
→561342→652341→564321 (P2)
123456→216453→162453→615423→561423→653421
→564321 (P2)
123456→214356→142356→415326→541326→456321
→564321 (P2)
123456→215436→126435→612435→163425→614325
→165324→651324→56$\underline{4}$321 (P3 type 2)

**P5:**

This pattern is just for the root group in Tree 3. This group comprises nodes whose last character is the same as that of the root. These nodes are attached to their parents (which belong to P4). This is illustrated by the following example.

123456→231456→325416→532416→351426→135426
123456→312456→135426
123456→216453→124653→412653→145623→514623
→153624→315624→134625→316425→135426(P5)
123456→214356→142356→413256→134256→315246
→132546→314526→135426
123456→215436→152436→513426→135426 ■



(a) Case 1: A triangle. (b) Case 2: Child backbone.

(c) Case 3: Cross node backbone.

**FIGURE 24.** P3 does not occur in backbones.

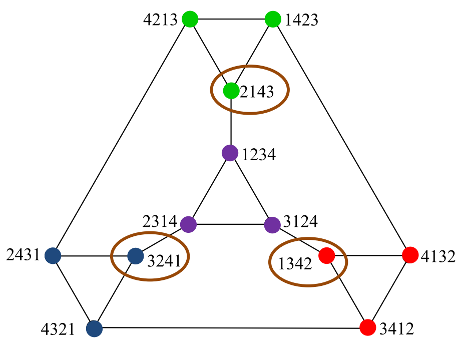


**FIGURE 25.** $AN_4$ group and bone seeds.

*Lemma 6:* For the path from the root to a node $v$ in group $A$, the backbone path of group $A$ and the spark parts of other groups will not have any common node except for $v$ and the root.

*Proof:* According to Lemma 5, the spark part has five patterns: P1, P2, P3, P4, and P5. For P1 and P2, assume that Tree $t$ is going to connect node $v$.

**P1:**

The third character of $v$ is identical to $t$ and node $v$ can be connected to Tree $t$ by a cross node operation directly. Therefore, no common node exists in the backbone path of Tree $A$.

**TABLE 3.** Bone seeds in $AN_{k+1}$.

| Tree id | Bone seed |
|---|---|
| 1 | $32(k+1)456...1$ |
| 2 | $13(k+1)456...2$ |
| 3 | $21(k+1)456...3$ |
| 5 | $12(k+1)436...5$ |
| 6 | $12(k+1)453...6$ |
| ... | ... |
| $k$ | $12(k+1)456...3k$ |

**TABLE 4.** Graph size and execution time.

| Size | $AN_3$ | $AN_4$ | $AN_5$ | $AN_6$ | $AN_7$ | $AN_8$ | $AN_9$ | $AN_{10}$ |
|---|---|---|---|---|---|---|---|---|
| Node amount | 3 | 12 | 60 | 360 | 2,520 | 20,160 | 181,440 | 1,814,400 |
| Edge amount | 3 | 18 | 120 | 900 | 7,560 | 70,560 | 725,760 | 8,164,800 |
| Time (seconds) | 0.01 | 0.02 | 0.04 | 0.15 | 1.03 | 10.38 | 121.39 | 1685.60 |

**TABLE 5.** $AN_3$ spark pattern.

| Tree 1 | Tree 2 |
|---|---|
| 2 | 1 |

**TABLE 6.** $AN_4$ spark pattern.

| Tree 1 | Tree 2 | Tree 3 |
|---|---|---|
| 2 | 1 | 1 |
| 2-2 | 1-1 | 1-1 |
| 3 | 3 | 1-1-4 |
| 3-3 | 3-3 | 2 |
| | | 2-2 |
| | | 2-2-4 |

**TABLE 7.** $AN_5$ spark pattern.

| Tree 1 | Tree 2 | Tree 3 | Tree 4 |
|---|---|---|---|
| 2 | 1 | 1 | 1 |
| 2-2 | 1-1 | 1-1 | 1-1 |
| 3 | 3 | 1-1-5 | 1-1-1 |
| 3-3 | 3-3 | 2 | 2 |
| 4 | 4 | 2-2 | 2-2 |
| 4-4 | 4-4 | 2-2-5 | 2-2-2 |
| 4-4-2 | 4-4-1 | 4 | 3 |
| 4-4-3 | 4-4-3 | 4-4 | 3-3 |
| | | 4-4-1 | 3-3-3 |
| | | 4-4-2 | 4 |
| | | 4-4-5 | 4-4 |
| | | 5 | 4-4-1 |
| | | | 4-4-1-1 |
| | | | 4-4-2 |
| | | | 4-4-2-2 |
| | | | 4-4-3 |
| | | | 4-4-3-3 |

**TABLE 8.** $AN_6$ spark pattern.

| Tree 1 | Tree 2 | Tree 3 | Tree 4 | Tree 5 |
|---|---|---|---|---|
| 2 | 1 | 1 | 1 | 1 |
| 2-2 | 1-1 | 1-1 | 1-1 | 1-1 |
| 2-2-2 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 |
| 3 | 3 | 1-1-6 | 2 | 2 |
| 3-3 | 3-3 | 2 | 2-2 | 2-2 |
| 3-3-3 | 3-3-3 | 2-2 | 2-2-2 | 2-2-2 |
| 4 | 4 | 2-2-2 | 3 | 3 |
| 4-4 | 4-4 | 2-2-6 | 3-3 | 3-3 |
| 4-4-5 | 4-4-3 | 4 | 3-3-3 | 3-3-3 |
| 5 | 4-4-5 | 4-4 | 4 | 4 |
| 5-5 | 5 | 4-4-1 | 4-4 | 4-4 |
| 5-5-5 | 5-5 | 4-4-2 | 4-4-1 | 4-4-1 |
| | 5-5-5 | 4-4-5 | 4-4-1-1 | 4-4-1-1 |
| | | 4-4-5-5 | 4-4-2 | 4-4-2 |
| | | 4-4-5-5-6 | 4-4-2-2 | 4-4-2-2 |
| | | 4-4-6 | 4-4-3 | 4-4-3 |
| | | 5 | 4-4-3-3 | 4-4-3-3 |
| | | 5-5 | 4-4-5 | |
| | | 5-5-5 | 4-4-5-5 | |
| | | 5-5-5-6 | 5 | |
| | | 5-5-6 | 5-5 | |
| | | 6 | 5-5-5 | |

**P2:**
Node $v$ is connected to Tree $t$ by a middle node $u$ in group $A$. Because edge $u \rightarrow v$ is not used in the backbone of Tree $A$ and node $u$ can be connected to Tree $t$ by a cross node operation directly, no common node exists in the backbone path of Tree $A$.

**P3:**
Assume that nodes are composed of "*abcde*..." and the tail part is omitted. The possibilities are exhausted by the following three cases.

**Case 1: a triangle**
Assume that node *bcade* requires a node whose third character is $c$ to reach the backbone of Tree $c$, as displayed in Figure 24(a). Because we use the TBFS process, there is no node like node *abcde*, which has only a left child, according to Lemma 2. Therefore, this case in which a collision occurs in node *abcde* will not exist.

**Case 2: child backbone**
As illustrated in Figure 24(b), node *cabde* in group $A$ requires the character $c$ to go to the backbone of Tree $c$, but $c$ is used by node *abcde*. Thus, node *cabde* uses the character $b$ to go to node *acdbe*. Subsequently, node *acdbe* goes to node *dacbe*, and finally, node *cabde* goes back to the backbone of Tree $c$. If a collision has occurred in node *dacbe* in the backbone of Tree $A$, the following path would occur: *abcde* $\rightarrow$ *badce* $\rightarrow$ *adbce* $\rightarrow$ *dacbe* (red lines). We use the TBFS process in bone functions. If the program has visited node *dacbe*, it will not use the red lines to visit node *cabde* during the backbone construction process. Instead, it will use the purple lines to visit node *cabde*. Therefore, the case does not occur because of the shortest path rule.

**Case 3: cross node backbone** As presented in Figure 24(c), node *acdbe* in group $A$ requires the character $b$ to go to the backbone of Tree $b$, but $b$ is used by node *cabde*. Thus, node *acdbe* uses $d$ to go to node *caebd*. Subsequently, node *caebd* goes to node *acbed*, and finally, node *acdbe* goes back to the backbone of Tree $b$. If a collision has occurred in node *acbed* in the backbone of Tree $A$, the following path would have existed: *acbed* $\rightarrow$ *caded* $\rightarrow$ *acedb* $\rightarrow$ *cabde* (red lines). We use the TBFS process in executing the bone functions. If the program has visited node *acbed*, it would not use the red lines and the blue line to visit node *acdbe* during the backbone construction process. Instead, it would

**TABLE 9.** *$AN_7$* **spark pattern.**

| Tree 1 | Tree 2 | Tree 3 | Tree 4 | Tree 5 | Tree 6 |
|---|---|---|---|---|---|
| 2 | 1 | 1 | 1 | 1 | 1 |
| 2-2 | 1-1 | 1-1 | 1-1 | 1-1 | 1-1 |
| 2-2-2 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 |
| 3 | 3 | 1-1-7 | 2 | 2 | 2 |
| 3-3 | 3-3 | 2 | 2-2 | 2-2 | 2-2 |
| 3-3-3 | 3-3-3 | 2-2 | 2-2-2 | 2-2-2 | 2-2-2 |
| 4 | 4 | 2-2-2 | 3 | 3 | 3 |
| 4-4 | 4-4 | 2-2-7 | 3-3 | 3-3 | 3-3 |
| 4-4-2 | 4-4-1 | 4 | 3-3-3 | 3-3-3 | 3-3-3 |
| 4-4-3 | 4-4-3 | 4-4 | 4 | 4 | 4 |
| 4-4-5 | 4-4-5 | 4-4-1 | 4-4 | 4-4 | 4-4 |
| 4-4-6 | 4-4-6 | 4-4-2 | 4-4-1 | 4-4-1 | 4-4-1 |
| 5 | 5 | 4-4-5 | 4-4-1-1 | 4-4-1-1 | 4-4-1-1 |
| 5-5 | 5-5 | 4-4-5-5 | 4-4-2 | 4-4-2 | 4-4-2 |
| 5-5-5 | 5-5-5 | 4-4-5-5-7 | 4-4-2-2 | 4-4-2-2 | 4-4-2-2 |
| 6 | 6 | 4-4-6 | 4-4-3 | 4-4-3 | 4-4-3 |
| 6-6 | 6-6 | 4-4-6-6 | 4-4-3-3 | 4-4-3-3 | 4-4-3-3 |
| 6-6-6 | 6-6-6 | 4-4-6-6-7 | 4-4-5 | 4-4-6 | 4-4-5 |
| | | 4-4-7 | 4-4-5-5 | 4-4-6-6 | 4-4-5-5 |
| | | 5 | 4-4-6 | 6 | 5 |
| | | 5-5 | 4-4-6-6 | 6-6 | 5-5 |
| | | 5-5-5 | 5 | 6-6-6 | 5-5-5 |
| | | 5-5-5-7 | 5-5 | | |
| | | 5-5-7 | 5-5-5 | | |
| | | 6 | 6 | | |
| | | 6-6 | 6-6 | | |
| | | 6-6-6 | 6-6-6 | | |
| | | 6-6-6-7 | | | |
| | | 6-6-7 | | | |
| | | 7 | | | |

use purple lines to visit node *acdbe*. Accordingly, the case will not occur because of the shortest path rule.

**P4:**

As illustrated in Figure 23, we has known that P2 can not occurs in the backbone and Tree 4 does not execute any bone operations. Therefore, no collision can exist in the backbone.

**P5:**

it is only used by the root group in Tree 3. Therefore, no collision occurs in the backbone. ■

*Lemma 7:* In Tree 3, the nodes of the root group must ultimately be connected.

*Proof:* Nodes in the root group belong to the previous trees. Because the incoming edges of the root group nodes are not used and Tree 3 does not have the previous tree, the root group nodes must ultimately be connected. For example, in $AN_6$, the paths of trees 1-5 from node 123456 to node 135426:

123456→231456→325416→532416→351426→135426
123456→312456 →135426
123456→216453→124653→412653→145623→514623
→153624→315624→134625→316425→135426
123456→214356→142356→413256→134256→315246
→132546→314526→135426
123456→215436→152436→513426→135426

Node 135426 in trees 1, 2, 4, and 5 belongs to the previous trees in $AN_5$. In tree 3, node 135426 must be connected ultimately. ■

*Theorem 1:* $n-1$ independent spanning trees in $AN_n$ constructed by the algorithm are independent.

**TABLE 10.** *$AN_8$* **spark pattern.**

| Tree 1 | Tree 2 | Tree 3 | Tree 4 | Tree 5 | Tree 6 | Tree 7 |
|---|---|---|---|---|---|---|
| 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2-2 | 1-1 | 1-1 | 1-1 | 1-1 | 1-1 | 1-1 |
| 2-2-2 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 |
| 3 | 3 | 1-1-8 | 2 | 2 | 2 | 2 |
| 3-3 | 3-3 | 2 | 2-2 | 2-2 | 2-2 | 2-2 |
| 3-3-3 | 3-3-3 | 2-2 | 2-2-2 | 2-2-2 | 2-2-2 | 2-2-2 |
| 4 | 4 | 2-2-2 | 3 | 3 | 3 | 3 |
| 4-4 | 4-4 | 2-2-8 | 3-3 | 3-3 | 3-3 | 3-3 |
| 4-4-2 | 4-4-1 | 4 | 3-3-3 | 3-3-3 | 3-3-3 | 3-3-3 |
| 4-4-3 | 4-4-3 | 4-4 | 4 | 4 | 4 | 4 |
| 4-4-5 | 4-4-5 | 4-4-1 | 4-4 | 4-4 | 4-4 | 4-4 |
| 4-4-6 | 4-4-6 | 4-4-2 | 4-4-1 | 4-4-1 | 4-4-1 | 4-4-1 |
| 4-4-7 | 4-4-7 | 4-4-5 | 4-4-1-1 | 4-4-1-1 | 4-4-1-1 | 4-4-1-1 |
| 5 | 5 | 4-4-5-5 | 4-4-2 | 4-4-2 | 4-4-2 | 4-4-2 |
| 5-5 | 5-5 | 4-4-5-5-8 | 4-4-2-2 | 4-4-2-2 | 4-4-2-2 | 4-4-2-2 |
| 5-5-5 | 5-5-5 | 4-4-6 | 4-4-3 | 4-4-3 | 4-4-3 | 4-4-3 |
| 6 | 6 | 4-4-6-6 | 4-4-3-3 | 4-4-3-3 | 4-4-3-3 | 4-4-3-3 |
| 6-6 | 6-6 | 4-4-6-6-8 | 4-4-5 | 4-4-6 | 4-4-5 | 4-4-5 |
| 6-6-6 | 6-6-6 | 4-4-7 | 4-4-5-5 | 4-4-6-6 | 4-4-5-5 | 4-4-5-5 |
| 7 | 7 | 4-4-7-7 | 4-4-6 | 4-4-7 | 4-4-7 | 4-4-6 |
| 7-7 | 7-7 | 4-4-7-7-8 | 4-4-6-6 | 4-4-7-7 | 4-4-7-7 | 4-4-6-6 |
| 7-7-7 | 7-7-7 | 4-4-8 | 4-4-7 | 6 | 5 | 5 |
| | | 5 | 4-4-7-7 | 6-6 | 5-5 | 5-5 |
| | | 5-5 | 5 | 6-6-6 | 5-5-5 | 5-5-5 |
| | | 5-5-5 | 5-5 | 7 | 7 | 6 |
| | | 5-5-5-8 | 5-5-5 | 7-7 | 7-7 | 6-6 |
| | | 5-5-8 | 6 | 7-7-7 | 7-7-7 | 7-7-7 |
| | | 6 | 6-6 | | | |
| | | 6-6 | 6-6-6 | | | |
| | | 6-6-6 | 7 | | | |
| | | 6-6-6-8 | 7-7 | | | |
| | | 6-6-8 | 7-7-7 | | | |
| | | 7 | | | | |
| | | 7-7 | | | | |
| | | 7-7-7 | | | | |
| | | 7-7-7-8 | | | | |
| | | 7-7-8 | | | | |
| | | 8 | | | | |

*Proof:* In the context of building small trees into larger trees, we use induction. If we know $AN_n$, then we know $AN_{n+1}$. When $n = 3$, basic part holds. There are two independent spanning trees, Tree 1 and Tree 2 as illustrated in Figure 8.

When $n = 4$, $AN_4$ is composed of four instances of $AN_3$. We divide nodes into groups by the last characters of each nodes.

First, we copy $T_1^3$ to $T_1^4$ and $T_2^3$ to $T_2^4$. Second, we find bone seeds (marked as ellipses with brown borders) as illustrated in Figure 25. Third, we use bone functions to create backbones. Fourth, we use spark functions to create spark parts. As displayed in Figure 10, since group 4 of $T_1^4$ and $T_2^4$ come from nodes of $T_1^3$ and $T_2^3$ with a character '4' appended in the end. The paths from the root to group 4 in $T_1^4$ and $T_2^4$ are internally vertex-disjoint. In $T_3^4$, group 4 are connected by group 1, 2, and 3. Therefore, for group 4, $T_1^4$, $T_2^4$, and $T_3^4$ are independent. For group 1, 2, and 3, $T_1^4$, $T_2^4$, and $T_3^4$ are also independent. Thus, $T_1^4$, $T_2^4$, and $T_3^4$ are independent.

Suppose that $AN_k$ holds. When $n = k + 1$, $AN_{k+1}$ is composed of $k + 1$ instances of $AN_k$. We divide nodes into groups according to the final character of each node. First, we copy $T_1^k$ to $T_1^{k+1}$, $T_2^k$ to $T_2^{k+1}$, $T_3^k$ to $T_k^{k+1}$, $T_4^k$ to $T_4^{k+1}$, $T_5^k$ to $T_5^{k+1}$, ... and $T_{k-1}^k$ to $T_{k-1}^{k+1}$. Second, we find bone seeds as explained in Table 3.

**TABLE 11.** $AN_9$ **spark pattern.**

| Tree 1 | Tree 2 | Tree 3 | Tree 4 | Tree 5 | Tree 6 | Tree 7 | Tree 8 |
|---|---|---|---|---|---|---|---|
| 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2-2 | 1-1 | 1-1 | 1-1 | 1-1 | 1-1 | 1-1 | 1-1 |
| 2-2-2 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 |
| 3 | 3 | 1-1-9 | 2 | 2 | 2 | 2 | 2 |
| 3-3 | 3-3 | 2 | 2-2 | 2-2 | 2-2 | 2-2 | 2-2 |
| 3-3-3 | 3-3-3 | 2-2 | 2-2-2 | 2-2-2 | 2-2-2 | 2-2-2 | 2-2-2 |
| 4 | 4 | 2-2-2 | 3 | 3 | 3 | 3 | 3 |
| 4-4 | 4-4 | 2-2-9 | 3-3 | 3-3 | 3-3 | 3-3 | 3-3 |
| 4-4-2 | 4-4-1 | 4 | 3-3-3 | 3-3-3 | 3-3-3 | 3-3-3 | 3-3-3 |
| 4-4-3 | 4-4-3 | 4-4 | 4 | 4 | 4 | 4 | 4 |
| 4-4-5 | 4-4-5 | 4-4-1 | 4-4 | 4-4 | 4-4 | 4-4 | 4-4 |
| 4-4-6 | 4-4-6 | 4-4-2 | 4-4-1 | 4-4-1 | 4-4-1 | 4-4-1 | 4-4-1 |
| 4-4-7 | 4-4-7 | 4-4-5 | 4-4-1-1 | 4-4-1-1 | 4-4-1-1 | 4-4-1-1 | 4-4-1-1 |
| 4-4-8 | 4-4-8 | 4-4-5-5 | 4-4-2 | 4-4-2 | 4-4-2 | 4-4-2 | 4-4-2 |
| 5 | 5 | 4-4-5-5-9 | 4-4-2-2 | 4-4-2-2 | 4-4-2-2 | 4-4-2-2 | 4-4-2-2 |
| 5-5 | 5-5 | 4-4-6 | 4-4-3 | 4-4-3 | 4-4-3 | 4-4-3 | 4-4-3 |
| 5-5-5 | 5-5-5 | 4-4-6-6 | 4-4-3-3 | 4-4-3-3 | 4-4-3-3 | 4-4-3-3 | 4-4-3-3 |
| 6 | 6 | 4-4-6-6-9 | 4-4-5 | 4-4-6 | 4-4-5 | 4-4-5 | 4-4-5 |
| 6-6 | 6-6 | 4-4-7 | 4-4-5-5 | 4-4-6-6 | 4-4-5-5 | 4-4-5-5 | 4-4-5-5 |
| 6-6-6 | 6-6-6 | 4-4-7-7 | 4-4-6 | 4-4-7 | 4-4-7 | 4-4-6 | 4-4-6 |
| 7 | 7 | 4-4-7-7-9 | 4-4-6-6 | 4-4-7-7 | 4-4-7-7 | 4-4-6-6 | 4-4-6-6 |
| 7-7 | 7-7 | 4-4-8 | 4-4-7 | 4-4-8 | 4-4-8 | 4-4-8 | 4-4-7 |
| 7-7-7 | 7-7-7 | 4-4-8-8 | 4-4-7-7 | 4-4-8-8 | 4-4-8-8 | 4-4-8-8 | 4-4-7-7 |
| 8 | 8 | 4-4-8-8-9 | 4-4-8 | 6 | 5 | 5 | 5 |
| 8-8 | 8-8 | 4-4-9 | 4-4-8-8 | 6-6 | 5-5 | 5-5 | 5-5 |
| 8-8-8 | 8-8-8 | 5 | 5 | 6-6-6 | 5-5-5 | 5-5-5 | 5-5-5 |
| | | 5-5 | 5-5 | 7 | 7 | 6 | 6 |
| | | 5-5-5 | 5-5-5 | 7-7 | 7-7 | 6-6 | 6-6 |
| | | 5-5-5-9 | 6 | 7-7-7 | 7-7-7 | 6-6-6 | 6-6-6 |
| | | 5-5-9 | 6-6 | 8 | 8 | 8 | 7 |
| | | 6 | 6-6-6 | 8-8 | 8-8 | 8-8 | 7-7 |
| | | 6-6 | 7 | 8-8-8 | 8-8-8 | 8-8-8 | 7-7-7 |
| | | 6-6-6 | 7-7 | | | | |
| | | 6-6-6-9 | 7-7-7 | | | | |
| | | 6-6-9 | 8 | | | | |
| | | 7 | 8-8 | | | | |
| | | 7-7 | 8-8-8 | | | | |
| | | 7-7-7 | | | | | |
| | | 7-7-7-9 | | | | | |
| | | 7-7-9 | | | | | |
| | | 8 | | | | | |
| | | 8-8 | | | | | |
| | | 8-8-8 | | | | | |
| | | 8-8-8-9 | | | | | |
| | | 8-8-9 | | | | | |
| | | 9 | | | | | |

**TABLE 12.** $AN_{10}$ **spark pattern (the root is 1234567890).**

| Tree 1 | Tree 2 | Tree 3 | Tree 4 | Tree 5 | Tree 6 | Tree 7 | Tree 8 | Tree 9 |
|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2-2 | 1-1 | 1-1 | 1-1 | 1-1 | 1-1 | 1-1 | 1-1 | 1-1 |
| 2-2-2 | 1-1-1 | 1-1-0 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 | 1-1-1 |
| 3 | 3 | 1-1-1 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3-3 | 3-3 | 2 | 2-2 | 2-2 | 2-2 | 2-2 | 2-2 | 2-2 |
| 3-3-3 | 3-3-3 | 2-2 | 2-2-2 | 2-2-2 | 2-2-2 | 2-2-2 | 2-2-2 | 2-2-2 |
| 4 | 4 | 2-2-0 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4-4 | 4-4 | 2-2-2 | 3-3 | 3-3 | 3-3 | 3-3 | 3-3 | 3-3 |
| 4-4-2 | 4-4-1 | 4 | 3-3-3 | 3-3-3 | 3-3-3 | 3-3-3 | 3-3-3 | 3-3-3 |
| 4-4-3 | 4-4-3 | 4-4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 4-4-5 | 4-4-5 | 4-4-0 | 4-4 | 4-4 | 4-4 | 4-4 | 4-4 | 4-4 |
| 4-4-6 | 4-4-6 | 4-4-1 | 4-4-1 | 4-4-1 | 4-4-1 | 4-4-1 | 4-4-1 | 4-4-1 |
| 4-4-7 | 4-4-7 | 4-4-2 | 4-4-1-1 | 4-4-1-1 | 4-4-1-1 | 4-4-1-1 | 4-4-1-1 | 4-4-1-1 |
| 4-4-8 | 4-4-8 | 4-4-5 | 4-4-2 | 4-4-2 | 4-4-2 | 4-4-2 | 4-4-2 | 4-4-2 |
| 4-4-9 | 4-4-9 | 4-4-5-5 | 4-4-2-2 | 4-4-2-2 | 4-4-2-2 | 4-4-2-2 | 4-4-2-2 | 4-4-2-2 |
| 5 | 5 | 4-4-5-5-0 | 4-4-3 | 4-4-3 | 4-4-3 | 4-4-3 | 4-4-3 | 4-4-3 |
| 5-5 | 5-5 | 4-4-6 | 4-4-3-3 | 4-4-3-3 | 4-4-3-3 | 4-4-3-3 | 4-4-3-3 | 4-4-3-3 |
| 5-5-5 | 5-5-5 | 4-4-6-6 | 4-4-5 | 4-4-6 | 4-4-5 | 4-4-5 | 4-4-5 | 4-4-5 |
| 6 | 6 | 4-4-6-6-0 | 4-4-5-5 | 4-4-6-6 | 4-4-5-5 | 4-4-5-5 | 4-4-5-5 | 4-4-5-5 |
| 6-6 | 6-6 | 4-4-7 | 4-4-6 | 4-4-7 | 4-4-7 | 4-4-6 | 4-4-6 | 4-4-6 |
| 6-6-6 | 6-6-6 | 4-4-7-7 | 4-4-6-6 | 4-4-7-7 | 4-4-7-7 | 4-4-6-6 | 4-4-6-6 | 4-4-6-6 |
| 7 | 7 | 4-4-7-7-0 | 4-4-7 | 4-4-8 | 4-4-8 | 4-4-8 | 4-4-7 | 4-4-7 |
| 7-7 | 7-7 | 4-4-8 | 4-4-7-7 | 4-4-8-8 | 4-4-8-8 | 4-4-8-8 | 4-4-7-7 | 4-4-7-7 |
| 7-7-7 | 7-7-7 | 4-4-8-8 | 4-4-8 | 4-4-9 | 4-4-9 | 4-4-9 | 4-4-9 | 4-4-8 |
| 8 | 8 | 4-4-8-8-0 | 4-4-8-8 | 4-4-9-9 | 4-4-9-9 | 4-4-9-9 | 4-4-9-9 | 4-4-8-8 |
| 8-8 | 8-8 | 4-4-9 | 4-4-9 | 6 | 5 | 5 | 5 | 5 |
| 8-8-8 | 8-8-8 | 4-4-9-9 | 4-4-9-9 | 6-6 | 5-5 | 5-5 | 5-5 | 5-5 |
| 9 | 9 | 4-4-9-9-0 | 5 | 6-6-6 | 5-5-5 | 5-5-5 | 5-5-5 | 5-5-5 |
| 9-9 | 9-9 | 5 | 5-5 | 7 | 7 | 6 | 6 | 6 |
| 9-9-9 | 9-9-9 | 5-5 | 5-5-5 | 7-7 | 7-7 | 6-6 | 6-6 | 6-6 |
| | | 5-5-0 | 6 | 7-7-7 | 7-7-7 | 6-6-6 | 6-6-6 | 6-6-6 |
| | | 5-5-5 | 6-6 | 8 | 8 | 8 | 7 | 7 |
| | | 5-5-5-0 | 6-6-6 | 8-8 | 8-8 | 8-8 | 7-7 | 7-7 |
| | | 6 | 7 | 8-8-8 | 8-8-8 | 8-8-8 | 7-7-7 | 7-7-7 |
| | | 6-6 | 7-7 | 9 | 9 | 9 | 9 | 8 |
| | | 6-6-0 | 7-7-7 | 9-9 | 9-9 | 9-9 | 9-9 | 8-8 |
| | | 6-6-6 | 8 | 9-9-9 | 9-9-9 | 9-9-9 | 9-9-9 | 8-8-8 |
| | | 6-6-6-0 | 8-8 | | | | | |
| | | 7 | 8-8-8 | | | | | |
| | | 7-7 | 9 | | | | | |
| | | 7-7-0 | 9-9 | | | | | |
| | | 7-7-7 | 9-9-9 | | | | | |
| | | 7-7-7-0 | | | | | | |
| | | 8 | | | | | | |
| | | 8-8 | | | | | | |
| | | 8-8-0 | | | | | | |
| | | 8-8-8 | | | | | | |
| | | 8-8-8-0 | | | | | | |
| | | 9 | | | | | | |
| | | 9-9 | | | | | | |
| | | 9-9-0 | | | | | | |
| | | 9-9-9 | | | | | | |
| | | 9-9-9-0 | | | | | | |
| | | 0 | | | | | | |

Third, we use the bone function to create backbones. Fourth, we use spark functions to create spark parts. For group $k+1$, all trees in $AN_k$ are independent and $T_3^{k+1}$ visits group $k+1$ through other groups; therefore, all paths from the root to group $k+1$ of all trees must be internally vertex-disjoint. For other groups, each path consists of backbone and spark parts. The backbone parts are different and the spark parts of all paths belong to one spark pattern of P1, P2, P3, P4, and P5, which does not have any node in common with the backbone part according to Lemma 6. Therefore, $k$ ISTs of $AN_{k+1}$ are independent. By induction hypothesis, the algorithm can construct $n-1$ ISTs in $AN_n$ accurately. ■

## V. IMPLEMENTATION

We program in PHP and create illustrations in Graphviz. From $AN_3$ to $AN_{10}$, we test whether any collision would occur. The results prove that no collision occurs. The algorithm is correct. We analyze the spark patterns as explained in the Appendix. The sizes of the graphs and the running times on a Dell R730 server are present in Table 4.

Because every node and every directed edge are traversed once, the time complexity of $AN_n$ is the summation of the numbers of nodes and directed edges from $AN_3$ to $AN_n$. The number of nodes in $AN_n$ is $\frac{n!}{2}$ and the number of directed edges in $AN_n$ is $\frac{n!}{2} \times (n-1)$. The time complexity is $O(n \times (\frac{n!}{2} + \frac{n!}{2} \times (n-1))) = O(n^2 \times n!)$.

## VI. CONCLUSION

In this paper, we propose a novel and simple top-down approach for creating independent spanning trees in alternating group networks. The main ideas of the algorithm are to use induction to develop small trees to big trees, and to use triangle and normal breadth-first search (BFS) processes, namely bone and park functions. Compared to other methods of different interconnection networks in the literature, the uniqueness of our method is that it does not need to determine the parent of one node by any rule, on the contrary others determine that by rules. The time complexity is polynomial time. We code this approach in PHP and test the implementation with inputs ranging from $AN_3$ to $AN_{10}$. The results prove that all trees of $AN_3$ to $AN_{10}$ are independent and that our algorithm is accurate and efficient.

## APPENDIX A
## SPARK PATTERN

See Tables 5–12.

## REFERENCES

[1] F. Bao, Y. Funyu, Y. Hamada, and Y. Igarashi, "Reliable broadcasting and secure distributing in channel networks," in *Proc. Int. Symp. Parallel Archit., Algorithms Netw. (I-SPAN)*, Taipei, Taiwan, Dec. 1997, pp. 472–478.

[2] A. A. Rescigno, "Vertex-disjoint spanning trees of the star network with applications to fault-tolerance and security," *Inf. Sci.*, vol. 137, nos. 1–4, pp. 259–276, Sep. 2001.

[3] Z. Hussain, B. AlBdaiwi, and A. Cerny, "Node-independent spanning trees in Gaussian networks," *J. Parallel Distrib. Comput.*, vol. 109, pp. 324–332, Nov. 2017.

[4] S.-Y. Hsieh and C.-J. Tu, "Constructing edge-disjoint spanning trees in locally twisted cubes," *Theor. Comput. Sci.*, vol. 410, nos. 8–10, pp. 926–932, Mar. 2009.

[5] Y.-J. Liu, J. K. Lan, W. Y. Chou, and C. Chen, "Constructing independent spanning trees for locally twisted cubes," *Theor. Comput. Sci.*, vol. 412, no. 22, pp. 2237–2252, May 2011.

[6] Y.-H. Chang, J.-S. Yang, S.-Y. Hsieh, J.-M. Chang, and Y.-L. Wang, "Construction independent spanning trees on locally twisted cubes in parallel," *J. Combinat. Optim.*, vol. 33, no. 3, pp. 956–967, Apr. 2017.

[7] J.-S. Yang, S.-M. Tang, J.-M. Chang, and Y.-L. Wang, "Parallel construction of optimal independent spanning trees on hypercubes," *Parallel Comput.*, vol. 33, no. 1, pp. 73–79, Feb. 2007.

[8] J. Werapun, S. Intakosum, and V. Boonjing, "An efficient parallel construction of optimal independent spanning trees on hypercubes," *J. Parallel Distrib. Comput.*, vol. 72, no. 12, pp. 1713–1724, Dec. 2012.

[9] J.-S. Yang, H.-C. Chan, and J.-M. Chang, "Broadcasting secure messages via optimal independent spanning trees in folded hypercubes," *Discrete Appl. Math.*, vol. 159, no. 12, pp. 1254–1263, Jul. 2011.

[10] S.-S. Kao, J.-M. Chang, K.-J. Pai, and R.-Y. Wu, "Constructing independent spanning trees on bubble-sort networks," in *Proc. Int. Comput. Combinatorics Conf. (COCOON)*, vol. 10976. Cham, Switzerland: Springer, 2018, pp. 1–13.

[11] S.-S. Kao, K.-J. Pai, S.-Y. Hsieh, R.-Y. Wu, and J.-M. Chang, "Amortized efficiency of constructing multiple independent spanning trees on bubble-sort networks," *J. Combinat. Optim.*, vol. 38, no. 3, pp. 972–986, Oct. 2019.

[12] S. B. Akers and B. Krishnamurthy, "A group-theoretic model for symmetric interconnection networks," *IEEE Trans. Comput.*, vol. 38, no. 4, pp. 555–566, Apr. 1989.

[13] Y. O. Hamidoune, A. S. Llado, and O. Serra, "The connectivity of hierarchical Cayley digraphs," *Discrete Appl. Math.*, vols. 37–38, pp. 275–280, Jul. 1992.

[14] S. Lakshmivarahan, J. S. Jwo, and S. K. Dhall, "Symmetry in interconnection networks based on Cayley graphs of permutation groups: A survey," *Parallel Comput.*, vol. 19, pp. 361–407, Apr. 1993.

[15] B. Chen, W. Xiao, and B. Parhami, "Internode distance and optimal routing in a class of alternating group networks," *IEEE Trans. Comput.*, vol. 55, no. 12, pp. 1645–1648, Dec. 2006.

[16] J.-S. Jwo, S. Lakshmivarahan, and S. K. Dhall, "A new class of interconnection networks based on the alternating group," *Networks*, vol. 23, no. 4, pp. 315–326, Jul. 1993.

[17] J. Youhu, "A new class of Cayley networks based on the alternating groups," (in Chinese), *Appl. Math.-J. Chin. Univ.*, vol. 14(A), no. 2, pp. 235–239, 1998.

[18] S. Zhou, W. Xiao, and B. Parhami, "Construction of vertex-disjoint paths in alternating group networks," *J. Supercomput.*, vol. 54, no. 2, pp. 206–228, Nov. 2010.

[19] A. Zehavi and A. Itai, "Three tree-paths," *J. Graph Theory*, vol. 13, no. 2, pp. 175–188, Jun. 1989.

[20] A. Itai and M. Rodeh, "The multi-tree approach to reliability in distributed networks," *Inf. Comput.*, vol. 79, no. 1, pp. 43–59, Oct. 1988.

[21] J. Cheriyan and S. N. Maheshwari, "Finding nonseparating induced cycles and independent spanning trees in 3-connected graphs," *J. Algorithms*, vol. 9, no. 4, pp. 507–537, Dec. 1988.

[22] S. Curran, O. Lee, and X. Yu, "Finding four independent trees," *SIAM J. Comput.*, vol. 35, no. 5, pp. 1023–1058, Jan. 2006.

[23] Y. Wang, J. Fan, G. Zhou, and X. Jia, "Independent spanning trees on twisted cubes," *J. Parallel Distrib. Comput.*, vol. 72, no. 1, pp. 58–69, Jan. 2012.

[24] B. Cheng, J. Fan, X. Jia, and S. Zhang, "Independent spanning trees in crossed cubes," *Inf. Sci.*, vol. 233, pp. 276–289, Jun. 2013.

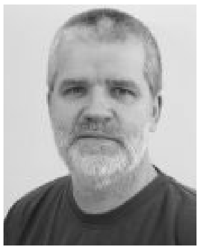
[25] B. Cheng, J. Fan, X. Jia, S. Zhang, and B. Chen, "Constructive algorithm of independent spanning trees on Möbius cubes," *Comput. J.*, vol. 56, no. 11, pp. 1347–1362, Nov. 2013.

[26] Y. Wang, J. Fan, X. Jia, and H. Huang, "An algorithm to construct independent spanning trees on parity cubes," *Theor. Comput. Sci.*, vol. 465, pp. 61–72, Dec. 2012.

**JIE-FU HUANG** received the B.S. degree from the Information Management Department, National Taiwan University, Taipei, Taiwan, in June 2003, and the M.S. degree from the Information Management Institute, National Cheng Kung University, Tainan, Taiwan, in June 2005. He is currently pursuing the Ph.D. degree with the Computer Science and Information Engineering Department, National Cheng Kung University. His current research interests include design and analysis of algorithms and graph theory.

**SHIH-SHUN KAO** received the B.S. degree from the Institute of Information and Decision Sciences, National Taipei University of Business, Taiwan, in 2018. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan, and the University of Bordeaux, France. His research interests include graph theory, interconnection networks, and algorithms.

**SUN-YUAN HSIEH** (Senior Member, IEEE) received the Ph.D. degree in computer science from National Taiwan University, Taipei, Taiwan, in June 1998. He served the compulsory two-year military service. From August 2000 to January 2002, he was an Assistant Professor with the Department of Computer Science and Information Engineering, National Chi Nan University. He joined the Department of Computer Science and Information Engineering, National Cheng Kung University, in February 2002, where he is currently a Distinguished Professor and the Dean of research. He is also with the Center for Innovative FinTech Business Models. His current research interests include design and analysis of algorithms, fault-tolerant computing, bioinformatics, parallel and distributed computing, and algorithmic graph theory. He is a Fellow of the British Computer Society (BCS). He received the 2007 K. T. Lee Research Award, the President's Citation Award (American Biographical Institute), in 2007, the Engineering Professor Award of Chinese Institute of Engineers (Kaohsiung Branch), in 2008, the National Science Council's Outstanding Research Award, in 2009, and the IEEE Outstanding Technical Achievement Award (IEEE Tainan Section), in 2011.



**RALF KLASING** received the Ph.D. degree from the Department of Mathematics and Computer Science, University of Paderborn, German, in 1995. From 2010 to 2015, he was the Head of the Combinatorics and Algorithms Team, LaBRI. He is currently a Senior Researcher (DR) with CNRS, LaBRI, Bordeaux. His research interests include distributed algorithms, approximation algorithms for combinatorially hard problems, algorithmic methods for telecommunication, and communication algorithms in networks.