IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received August 16, 2020, accepted September 7, 2020, date of publication September 18, 2020, date of current version October 6, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3025009

Two Algorithms for Constructing Independent Spanning Trees in (*n*,*k*)-Star Graphs

JIE-FU HUANG[1], EDDIE CHENG[2], AND SUN-YUAN HSIEH[3], (Senior Member, IEEE)
[1]Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan 701, Taiwan
[2]Department of Mathematics and Statistics, Oakland University, MI 48309, USA
[3]Department of Computer Science and Information Engineering, Institute of Medical Informatics, Institute of Manufacturing Information and Systems, International Center for the Scientific Development of Shrimp Aquaculture, Center for Innovative FinTech Business Models, National Cheng Kung University, Tainan 701, Taiwan

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

ABSTRACT In a graph G, two spanning trees T_1 and T_2 are rooted at the same vertex r. If, for every $v \in V(G)$, the paths from v to the root r in T_1 and T_2 are internally vertex-disjoint, they are called independent spanning trees (ISTs). ISTs can be applied in numerous fields, such as fault-tolerant broadcasting and secure message distribution. The (n,k)-star graphs $S_{n,k}$ constitute a generalized version of the star network. The method of constructing ISTs in (n,k)-star graphs remains unknown. In this paper, we propose one recursive algorithm and one parallel algorithm for constructing ISTs on (n,k)-star graphs. The main ideas of the recursive algorithm are to use induction to change small trees into large trees, to use a modified breadth-first search (MBFS) traversal to create a frame for an IST, and to use a breadth-first search (BFS) traversal to connect the rest of nodes. The main ideas of the parallel algorithm are to create frames through MBFS traversals in parallel, and to use some specific rules to connect the rest of nodes in parallel. We also present validation proofs for both algorithms, and analyze the time complexities of both algorithms. The time complexity of the recursive algorithm in $S_{n,k}$ is $O(n \times \frac{n!}{(n-k)!})$, where $\frac{n!}{(n-k)!}$ is the number of nodes of $S_{n,k}$. The time complexity of the parallel algorithm can be reduced to $O(\frac{n!}{(n-k)!})$ if the system has $n-1$ processors computing in parallel. Both algorithms are correct with the stated time complexity values; the parallel algorithm is more efficient than the recursive algorithm.

INDEX TERMS Independent spanning trees, (n,k)-star graphs, breadth-first search, recursive algorithm, parallel algorithm.

I. INTRODUCTION

In a graph G, two spanning trees T_1 and T_2 are rooted at the same vertex r. If, for every $v \in V(G)$, the paths from v to the root r in T_1 and T_2 are internally vertex-disjoint, they are called independent spanning trees (ISTs). ISTs can be applied in numerous fields, such as fault-tolerant broadcasting and secure message distribution [1], [22]. Assume that a network N has k ISTs rooted at node r, and N contains at most $k-1$ faulty nodes. These applications are illustrated as follows [17]:

- In fault-tolerant broadcasting, node r broadcasts a message to every non-faulty node v in N through the k ISTs. Because 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 promises the message can be delivered to every node in N reliably;
- In secure message distribution, node r divides a message into k packets, and sends each packet to the destination through a different IST. Thus, each node in N receives at most one of the k packets except the destination node, which receives all k packets.

Scholars have described how to construct ISTs in graphs. Zehavi and Itai proposed a conjecture that there exist k ISTs in a k-connected graph [28]. The conjecture has been proven correct for k-connected graphs such that $k \leq 4$ [9], [12], [18], but it is still unknown for graphs such that $k \geq 5$. However, the problem is considerably challenging for arbitrary graphs, and from 2007 to 2020, researchers have published studies on specifying ISTs in interconnection networks.

The constructions of ISTs on several interconnection networks have been solved, including crossed cubes [7], Möbius cubes [8], twisted cubes [24], locally twisted cubes [13], [21],

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

hypercubes [26], parity cubes [23], folded hypercubes [27], and bubble-sort networks [19], [20], alternating group networks [16]. In those studies, some parallel algorithms have been proposed on twisted cubes [5], locally twisted cubes [4], parity cubes [3], and hypercubes [25], [26]. To the best of our knowledge, the method of constructing ISTs in (n,k)-star graphs is still unknown. In this paper, we concentrate on (n,k)-star graphs.

An (n, k)-star graph $S_{n,k}$ refers to a generalized version of an n-star graph S_n, where $S_{n,n-1}$ and S_n are isomorphic and $S_{n,1}$ is obviously a complete graph K_n. Scholars have computed or derived some basic properties of graphs of the form $S_{n,k}$, such as diameter [10], connectivity [10], broadcasting [6], average distance [11], embedding [2], Hamiltonicity [14], spanning connectivity [15], and wide diameter [15]. These results demonstrate that $S_{n,k}$ graphs have excellent topological properties.

In this paper, we propose one recursive algorithm and one parallel algorithm. Both of which construct ISTs in (n,k)-star graphs. The main ideas of the recursive algorithm are to use induction to extend small trees into big trees, use a modified breadth-first search (MBFS) traversal to create a frame of an IST, and use a breadth-first search (BFS) traversal to connect the remaining nodes. The main ideas of the parallel algorithm are to create frames through MBFS traversals in parallel and use the specific rules to connect the rest of nodes in parallel. In this paper, we validate both algorithms, and analyze the time complexity of both algorithms. The time complexity of the recursive algorithm in $S_{n,k}$ is $O(n \times \frac{n!}{(n-k)!})$, where $\frac{n!}{(n-k)!}$ is the number of nodes of $S_{n,k}$, whereas that of the parallel algorithm can decline to $O(\frac{n!}{(n-k)!})$ if the system has $n-1$ processors computing in parallel. Both algorithms are correct with the stated time complexity; furthermore, the parallel algorithm is more efficient than the recursive algorithm.

The remainder of this paper is organized as follows: Section II explains preliminary considerations, Section III presents both algorithms, Section IV proves the relevant claims, Section V analyzes and compares both algorithms, and Section VI concludes the paper.

II. PRELIMINARY CONSIDERATIONS

Let $\langle n \rangle = \{1, 2, \ldots, n\}$ and $\langle k \rangle = \{1, 2, \ldots, k\}$.

Definition 1 ([10]): An (n, k)-star graph, denoted by $S_{n,k}$, is specified by two integers n and k, where $1 \le k < n$. The node set of $S_{n,k}$ is denoted by $\{p_1p_2 \ldots p_k \mid p_i \in \langle n \rangle \text{ and } p_i \neq p_j \text{ for } i \neq j\}$. The adjacency is defined as follows: $p_1p_2 \ldots p_i \ldots p_k$ is adjacent to

(1) $p_ip_2 \ldots p_{i-1}p_1p_{i+1} \ldots p_k$ through an edge of dimension i, where $2 \le i \le k$ (swap p_1 and p_i) and

(2) $xp_2 \ldots p_k$ through dimension 1, where $x \in \langle n \rangle - \{p_i \mid 1 \le i \le k\}$.

The node of type (1) is referred to as friend i of $p_1p_2 \ldots p_i \ldots p_k$, and the node of type (2) is referred to as child z of $p_1p_2 \ldots p_i \ldots p_k$, where z is the ordinal number of x in $\langle n \rangle - \{p_i \mid 1 \le i \le k\}$ in ascending order.

$S_{3,1}$ is depicted in Figure 1. Node 3 has two children, namely node 1 (child 1) and node 2 (child 2). $S_{4,2}$ is displayed in Figure 2. Node 34 has two children and one friend as presented in Figure 3. $S_{5,3}$ consists of 60 nodes and 120 edges, as displayed in Figure 4, and is composed of five instances $S_{4,2}$.

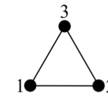


FIGURE 1. $S_{3,1}$.

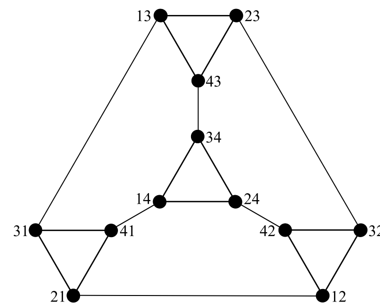
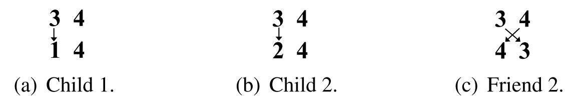


FIGURE 2. $S_{4,2}$.



(a) Child 1. (b) Child 2. (c) Friend 2.

FIGURE 3. Children and friend of node 34 in $S_{4,2}$.

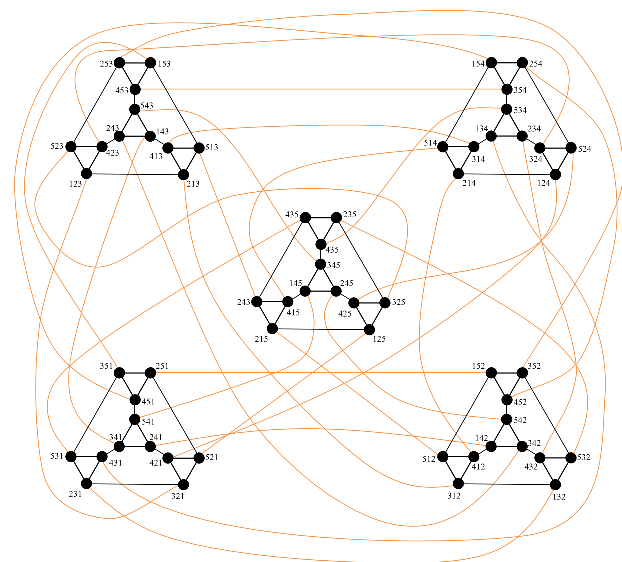


FIGURE 4. $S_{5,3}$.

Like the n-star graph S_n, an $S_{n,k}$ can be decomposed into n instances of $S_{n-1,k-1}$. That is, an $S_{n,k}$ can be decomposed into n vertex-disjoint instances of $S_{n-1,k-1}$ in $k-1$ different ways by fixing the symbol in any position i, such

that $2 \leq i \leq k$. This decomposition can be conducted recursively on each $S_{n-1,k-1}$ to obtain smaller subgraphs.

Lemma 1 ([10]): (n, k)-star graphs is vertex symmetric. According to Lemma 1, any node in $S_{n,k}$ can be the root in the process of constructing ISTs.

Lemma 2 ([10]): $(n, n-1)$-star graph $S_{n,n-1}$ is isomorphic to the n-star graph S_n

III. ALGORITHMS

Notations used here are defined as follows:

- the root: node $(n-k+1)(n-k+2)(n-k+3)\ldots n$;
- cluster A: a set of nodes whose last symbol is A;
- the root cluster: the set of nodes whose last symbol is identical to that of the root. In $S_{n,k}$, the root cluster is cluster n;
- frame: a set of nodes with the same last symbols created by buildFrame function, which is defined later in a later passage;
- germ: the starting node of a frame;
- $T_j^{n,k}$: the jth IST of $S_{n,k}$, and the last symbol of its frame is j;
- tid: the last symbol of the germ, namely j of $T_j^{n,k}$.

$S_{n,k}$ is partitioned into n instances of $S_{n-1,k-1}$. Every node is classified into a cluster by its last symbol. Thus, $S_{n,k}$ has n clusters. For example, $S_{4,2}$ can be divided into four instances of $S_{3,1}$. $S_{4,2}$ has four clusters, as illustrated in Figure 5.

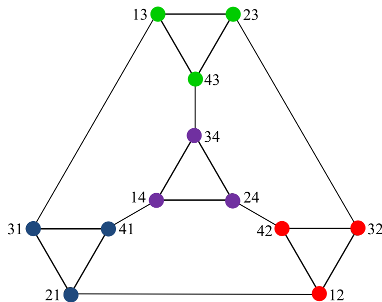


FIGURE 5. Four clusters of $S_{4,2}$.

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

In $S_{n,k}$, the root is incident to $n-1$ edges and adjacent to $n-1$ nodes. Considering the $n-1$ nodes adjacent to the root, $n-2$ nodes can be constructed from the nodes in $S_{n-1,k-1}$ if one symbol n is appended to the kth position, but the friend k is a new node. Each of the $n-2$ nodes except for friend 2 has a new edge that is adjacent to a node that is called a **germ** and the friend k itself is also a germ. We use germs to create a frame to connect other clusters.

The concept is illustrated in Figure 6. Each of the central white nodes can be formed from some node in $S_{n-1,k-1}$ if a single symbol n is appended to the kth position.

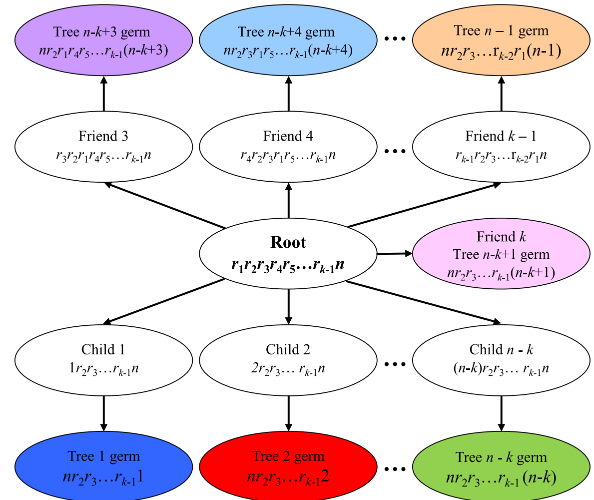


FIGURE 6. Germs ($r_i = n - k + i$; hence, $r_k = n$).

Child 1 of the root in $S_{n,k}$ uses its new edge to make the germ of $T_1^{n,k}$. Child 2 of the root in $S_{n,k}$ uses its new edges to make the germ of $T_2^{n,k}$. The root in $S_{n,k}$ uses its new edge to make the germ of $T_{n-k+1}^{n,k}$, namely friend k. Friends 3 to $k-1$ of the root in $S_{n,k}$ use their new edges to make the germs of $T_{n-k+3}^{n,k}$ to $T_{n-1}^{n,k}$. $T_{n-k+2}^{n,k}$ does not conduct any buildFrame function; consequently, it has no germ, and that is why we skip friend 2 in Figure 6.

Basic variable and functions:

- $tary$: an array of ISTs of this iteration; it is a two-dimensional array, and $tary[tid][node]$ stores its parent. Therefore, every IST includes all nodes. If the node has not yet been traversed, $tary[tid][node] = -1$.
- $ptary$: an array of the trees of the previous iteration.
- $dedge$: an associative array for storing all direct edges only used in the recursive algorithm. Initially, a $dedge[from][to] = 1$ means an unused direct edge; a $dedge[from][to] = 0$ means a used direct edge.
- function getChild($parent$): returns the children of $parent$.
- function getFriend($parent$, p): returns the friend p of $parent$ by swapping the first and the pth symbols of $parent$.
- function checkChild($tary$, tid, $dedge$, $parent$, $children$): returns true if any child of $parent$ in $children$ is unvisited in Tree tid, false otherwise. $dedge$ is not required and set to false in the parallel algorithm.

A. buildFrame FUNCTION

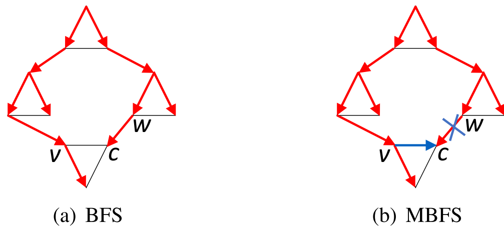
A frame tree can be constructed from a germ through a modified BFS (MBFS) traversal. However, the MBFS traversal must stop when it encounters a node of which the last symbol differs from that of the germ.

BFS order. In the buildFrame function, each node in $frQue$ must traverse all unvisited nodes connected to it in this order:

1. child 1,2, 3 …, $n-k$;
2. friend 2, 3, 4, …, $k-1$.

```
Function checkChild(tary, tid, dedge, parent, children)
  Input  : tary, tid, dedge, parent, children
  Output: true or false
1 for each child c in children do
2     if dedge == false then
3         if tree[tid][c] == −1 then
4             return true;
5     else
6         if tree[tid][c] == −1 and
          dedge[parent][c] == 1 then
7             return true;
8 return false;
```

MBFS traversal. Because $S_{n,k}$ is symmetric, some child c of one node v may be traversed by another node w earlier. In such case, we should transfer c's parent from w to v. For instance, in $S_{n,n-2}$, the graph presented in Figure 7(a) occurs. We set the edge (marked with a blue X) from used to unused and use another edge (marked with a blue arrow) to traverse c. Hence, the triangle shape is retained. Figure 7(b) illustrates this idea.



(a) BFS (b) MBFS

FIGURE 7. BFS and MBFS traversals.

For example, in Figures 8, 9, and 10, the frame nodes of $T_1^{5,3}$, $T_2^{5,3}$, and $T_3^{5,3}$ are indicated by ellipses with blue borders.

B. BuildLeaf FUNCTION

The buildLeaf function connects all unvisited nodes one step at a time from a tree after the buildFrame function. For example, in Figure 11, the leaf nodes of $T_4^{5,3}$ in the first, second, third, and fourth buildLeaf executions are indicated by ellipses with brown, gold, gray, and red borders, respectively.

C. RECURSIVE ALGORITHM

The recursive algorithm is presented in Algorithm 1.

$S_{n-k+1,1}$, **the basic part,** is a complete graph, having $n-k$ independent spanning trees that are illustrated in Figure 12.

Copy previous trees. At the end of each iteration, the trees that had been used must be stored in a tree array called *ptary* so that correct information can be used in the next iteration. At the start of every iteration, the program should copy previous trees to the tree array *tary*. Table 1 presents the mapping

```
Function buildFrame(tary, tid, dedge, germ)
  Input  : tary, tid, germ
  Output: the frame of tary[tid]
1  frQue = array(germ);                ▷ Put germ into frQue.
2  for j = 0; j < |frQue|; j = j + 1 do
3      v = frQue[j];             ▷ the (j + 1)th element of frQue
4      children = getChild(v);
5      // any child unvisited
6      if checkChild(tary, tid, dedge, v, children) then
7          for each element c in children do
8              if c has been visited then
9                  remove c from frQue;
10                 dedge[tary[tid][c]][c]++;
11                 // Set the edge (from c's parent to c)
12                 // unused (from 0 to 1).
13             // visit child c
14             tary[tid][c] = v;                         ▷ (1)
15             dedge[v][c] = 0;                          ▷ (2)
16             frQue[] = c;                              ▷ (3)
17     // visit friend 2 to |germ| − 1
18     for i = 2; i ≤ the number of symbols of germ − 1;
       i = i + 1 do
19         fd = getFriend(v, i);
20         if tary[tid][fd] == 0 and dedge[v][fd] == 1
           then
21             tary[tid][fd] = v;                        ▷ (1)
22             dedge[v][fd] = 0;                         ▷ (2)
23             frQue[] = fd;                             ▷ (3)
24 // (1)Set its parent.
25 // (2)Set the edge used (from 1 to 0).
26 // (3)Put it into frQue.
27 // Red text indicates a portion that differs from the
28 // buildFrameP function.
```

TABLE 1. Tree id and previous tree id mapping.

Tree id	1	2	...	$n-k$	$n-k+1$	$n-k+2$	...	$n-1$
Previous tree id	1	2	...	$n-k$	no previous tid, a new tree	$n-k+2$	...	$n-k+1$

a new tree and an old tree. In a typical iteration, $T_1^{n,k}$ copies information from $T_1^{n-1,k-1}$ and $T_{n-1}^{n,k}$ copies information from $T_{n-k+1}^{n-1,k-1}$. $T_{n-k+1}^{n,k}$ is a new tree, and thus no previous tree is copied.

Create a framelist. A *framelist* is an array for storing the germs. The germs are created from the root. In $S_{4,2}$, we can create three germs, namely 41, 42, and 43 to establish frames, as illustrated in Figure 13.

For example, $S_{5,3}$ has four trees and three germs: 541 in $T_1^{5,3}$, 542 in $T_2^{5,3}$, and 543 in $T_3^{5,3}$, as presented Figures 8, 9, and 10, respectively. As indicated in Figure 11, $T_4^{5,3}$ does not execute any buildFrame function, but it does conduct buildLeaf functions.

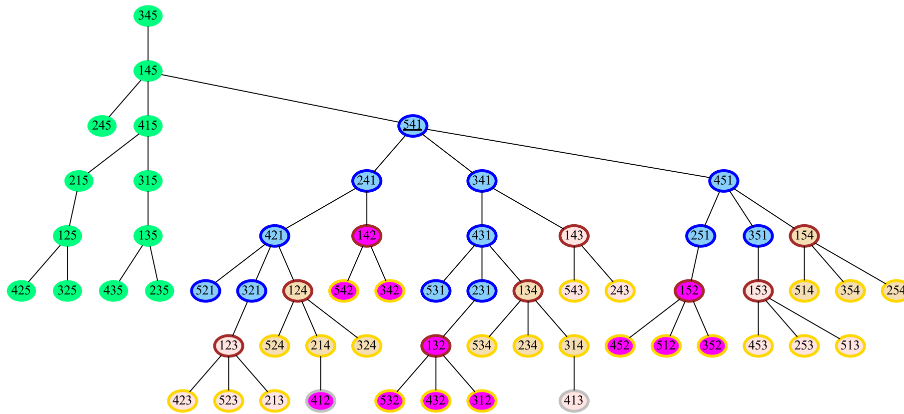


FIGURE 8. $T_1^{5,3}$ **germ: 541 (underlined). Frame nodes (blue border), and leaf nodes (brown, gold, and gray border).**

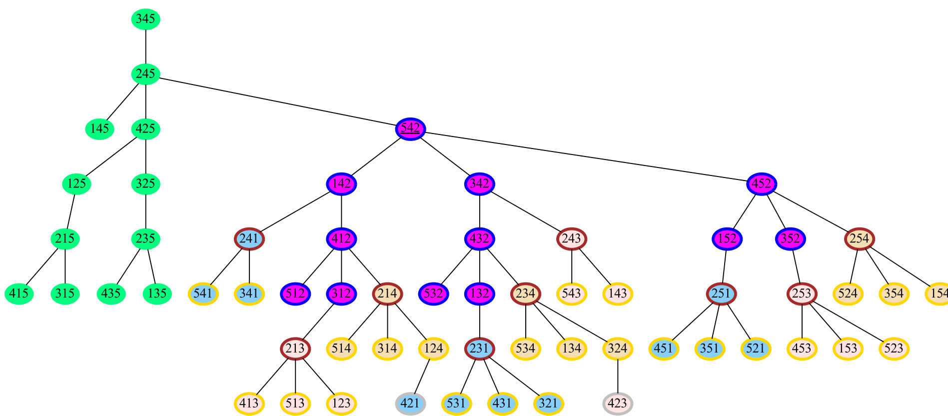


FIGURE 9. $T_2^{5,3}$ **germ: 542 (underlined). Frame nodes (blue border), and leaf nodes (brown, gold, and gray border).**

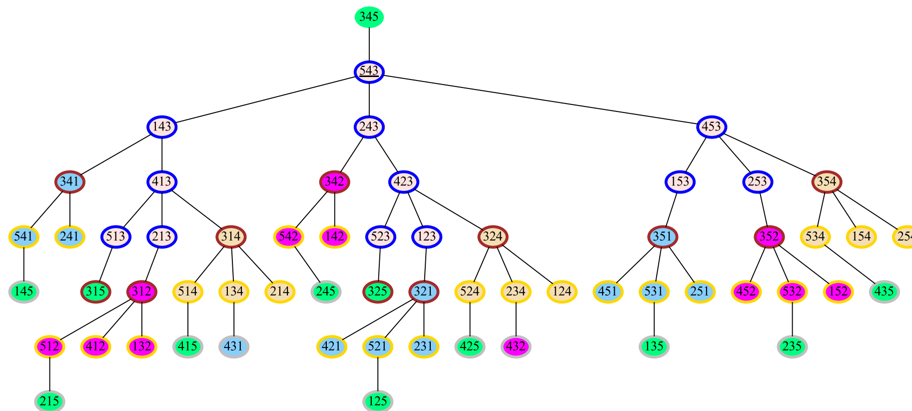


FIGURE 10. $T_3^{5,3}$ **germ: 543 (underlined). Frame nodes (blue border), and leaf nodes (brown, gold, and gray border).**

D. *BuildFrameP FUNCTION*

The frame of an IST comprises nodes of the same tail portion within this IST. The frame of an IST can be constructed from a germ through a modified breadth-first search (MBFS) traversal displayed in Figure 7. However, the MBFS traversal must stop when it encounters a node of which the last *tailN* symbol(s) differ(s) from that of the germ.

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

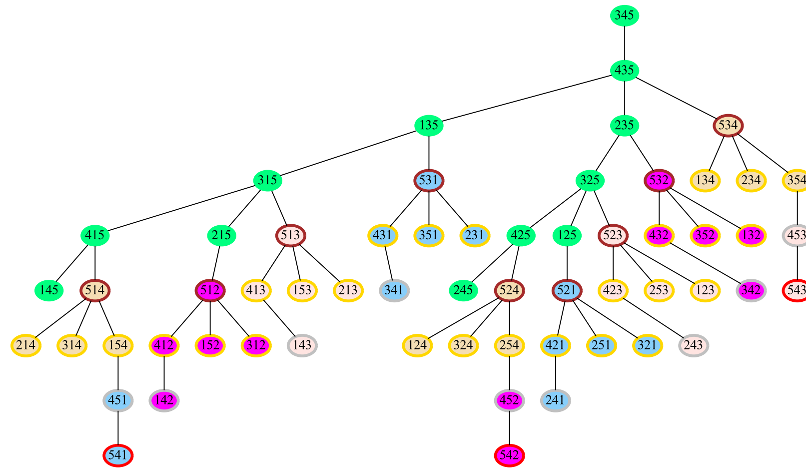


FIGURE 11. $T_4^{5,3}$. **No germ exists no buildFrame operation exists. Leaf nodes (brown, gold, gray, and red border).**

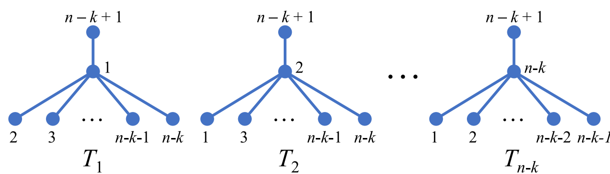


FIGURE 12. $n-k$ **ISTs of** $S_{n-k+1,1}$.

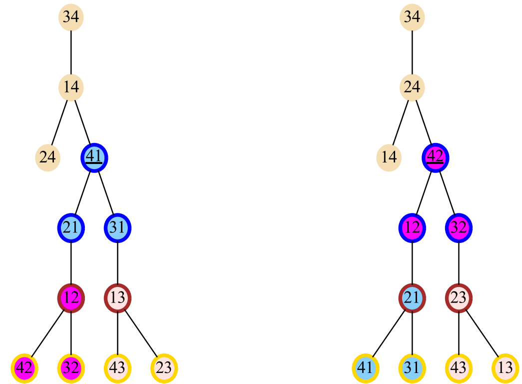
1. child 1,2, 3 …, $n-k$;
2. friend 2, 3, 4, …, $k-tailN$.

In Figure 14, 15, 16, 17, and 18, the frame nodes are underlined. The buildFrameP function will not encounter any node whose last *tailN* symbol(s) is(are) different from that of the germ because it does not visit the last *tailN* friend(s).

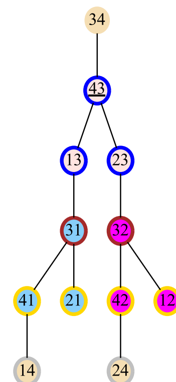
E. FindParent FUNCTION

Let $Frame_j^{n,k}$ denote that the frame of $T_j^{n,k}$. In $T_j^{n,k}$ construction, each unvisited node *child* whose last symbol is z should be connected to $Frame_j^{n,k}$ after the buildFrameP functions. The findParent function is used to find the parent of *child* in $T_j^{n,k}$ by means of the findTarget function according to the following rules.

- P1 rule:
  If *child*'s first symbol is j, the findTarget function swaps the first and last symbols of *child* to return the target node;
- P2 rule:
  If *child*'s first symbol is not j, the findTarget function moves j to first position of *child* by child, or friend operation to return the target node;
- P3 rule:
  If the target node has been used by $Frame_z^{n,k}$, the target is changed to $n-k+2$, and the findTarget function is used again to move $n-k+2$ to the first or last position of *child* to return another target node;
- T_{n-k+1} rule:
  If *child* is in the root cluster and j is $n-k+1$, the findParent function swaps the first and last symbols of *child* to form its parent.



(a) $T_1^{4,2}$ germ: 41. (b) $T_2^{4,2}$ germ: 42.



Germs (underlined).
Frame nodes (blue border).
Leaf nodes (brown, gold, and gray border)

(c) $T_3^{4,2}$ germ: 43.

FIGURE 13. **Germs, frame nodes, and leaf nodes in** $S_{4,2}$.

Function buildLeaf(*tary*, *tid*, *dedge*, *first*, *n*, *k*)

```
Input  : tary, tid, dedge, first, n, k
Output: nodes can be visited by tary[tid] in a step
1  // global means using global variable
2  global lfQue;          ▷ a two-dimensional array storing
3     ▷ nodes visited by T_tid not to visit all nodes next time
4  global lfQIx;          ▷ the starting position in lfQue[tid]
5  if first == true then
6      // first execution
7      for every node v in tary[tid] do
8          //only T^{n,k}_{n-k+2} can grow from the root cluster
9          if v has parent and ( ( tid ≠ n − k + 2 and the
           last symbol of v ≠ the last symbol of the root) or
           tid == n − k + 2) then
10             children = getChild(v);
11             for each element c in children do
12                 // visit child c
13                 if tary[tid][c] == 0 and
                   dedge[v][c] == 1 then
14                     tary[tid][c] = v; dedge[v][c] = 0;
15                     lfQue[tid][] = c;          ▷ (1)(2)(3)
16             // visit the friend 2 to k
17             for i = 2; i ≤ k; i = i + 1 do
18                 fd = getFriend(v, i);
19                 if tary[tid][fd] == 0 and
                   dedge[v][fd] == 1 then
20                     tary[tid][fd]=v; dedge[v][fd]=0;
21                     lfQue[tid][] = fd;         ▷ (1)(2)(3)
22 else
23     qs = count(lfQue[tid]);      ▷ the size of lfQue[tid]
24     // New element will be put into lfQue[tid], but
25     // will be checked in next execution.
26     for y = lfQIx[tid]; y < qs; y = y + 1 do
27         v = lfQue[tid][y];
28         children = getChild(v);
29         for each element c in children do
30             // visit child c
31             if tary[tid][c] == 0 and dedge[v][c] == 1
               then
32                 tary[tid][c] = v; dedge[v][c] = 0;
33                 lfQue[tid][] = c;              ▷ (1)(2)(3)
34         // visit the friend 2 to k
35         for i = 2; i ≤ k; i = i + 1 do
36             fd = getFriend(v, i);
37             if tary[tid][fd] == 0 and
               dedge[v][fd] == 1 then
38                 tary[tid][fd] = v; dedge[v][fd] = 0;
39                 lfQue[tid][] = fd;             ▷ (1)(2)(3)
40     lfQIx[tid] = y;        ▷ the position for next execution
41 // (1)Set its parent. (2)Set the edge used (from 1 to 0).
42 // (3)Put it into lfQue[tid].
```

Algorithm 1 Recursive Algorithm

```
Input  : n and k                  ▷ the dimensions of S_{n,k}
Output: the ISTs from S_{n-k+1,1} to S_{n,k}
1  for t = n − k + 1; t ≤ n; t = t + 1 do
2      if t == n − k + 1 then
3          basic part, a complete graph;
4      else if t ≥ n − k + 2 then
5          Copy previous trees of the previous iteration to
           the trees of this iteration;
6          Create a framelist;      ▷ an array storing germs
7          for every germ in the framelist do
8              Execute buildFrame function;          ▷ (1)
9          // buildLeaf block
10         if t == n − k + 2 then
11             // S_{n-k+2,2} executes buildLeaf function
12             // three times.
13             for i = 0; i < 3; i = i + 1 do
14                 for j = 1; j ≤ t − 1; j = j + 1 do
15                     if i == 0 then
16                         buildLeaf(tary, j, dedge, true, n,
                           k);
17                     else
18                         buildLeaf(tary, j, dedge, false, n,
                           k);
19         else if t ≥ n − k + 3 then
20             // Every tree executes buildLeaf function
21             // four times.
22             for i = 0; i < 4; i = i + 1 do
23                 for j = 1; j ≤ t − 1; j = j + 1 do
24                     if i == 0 then
25                         buildLeaf(tary, j, dedge, true, n,
                           k);
26                     else
27                         buildLeaf(tary, j, dedge, false, n,
                           k);
28             buildLeaf(tary, n − k + 1, dedge, true, n, k);
               ▷ (2)
29     ptary = tary;
30             ▷ tary must be stored to use in next iteration.
31 /* Comments:
32 (1) Tree n − k + 2 executes a buildLeaf function to
   produce friends along the root cluster, Tree n − k + 2
   first occurs in S_{n-k+3,3}, and it copies Tree n − k + 1 of
   S_{n-k+2,2} to its own tree array, namely tary[n − k + 2].
33 (2) Relative to the other trees, Tree n − k + 1 needs one
   more buildLeaf function to connect the root cluster. */
```

Basic function:

- substring(*string*, *start*, *length*): returns *length* symbols of *string* from the *start*th position. Suppose the position

```
Function buildFrameP(tary, tid, germ, tailN)
  Input : tary, tid, germ, tailN
  Output: the frame of tary[tid] whose last tailN
          symbol(s) is(are) fixed
1  frQue=array(germ);                    ▷ Put germ into frQue
2                                        ▷ initially.
3  for j = 0; j < |frQue|; j = j + 1 do
4      v = frQue[j];          ▷ the (j + 1)th element of frQue.
5      children = getChild(v);
6      // any child unvisited
7      if checkChild(tary, tid, false, v, children) then
8          for each element c in children do
9              if c has been visited then
10                 remove c from frQue;
11             // visit child c
12             tary[tid][c] = v;               ▷ Set c's parent.
13             frQue[] = c;                    ▷ Put c into frQue.
14     // visit the friend 2 to |germ| − tailN
15     for i = 2; i ≤ the number of symbols of
       germ − tailN; i = i + 1 do
16         fd = getFriend(v, i);
17         if tary[tid][fd] == 0 then
18             tary[tid][fd] = v;              ▷ Set fd's parent.
19             frQue[] = fd;                   ▷ Put fd into frQue.
20     // Red text indicates a portion that differs from the
21     // buildFrame function.
```

```
Function findTarget(child, target)
  Input : child, target
  Output: child's target node
1  p = the position of target in child;
2  if p == false then
3      return target+substring(child, 2, the length of child -
       1);                             ▷ target is not in child.
4  else if p == 1 then
5      // Swap the first and last symbols of child.
6      return getFriend(child, the length of child);
7                          ▷ P1 or P3 rule (target is n − k + 2).
8  else
9      return getFriend(child, p);
10                 ▷ Swap the first and pth symbols of child.
```

starts at 1. If $start > |string|$ or $length = 0$, an empty string is returned.

If the last symbol of *child* is identical to the dimension n of $S_{n,k}$, we can reduce the dimension until the last symbol of *child* is not identical to the dimension n of $S_{n,k}$. According to Table 2, the $(n-k+1)$th IST in $S_{x,x-n+k}$ is part of the last IST $S_{x+1,x+1-n+k}$. Namely $T^{x,x-n+k}_{n-k+1}$ is used by $T^{x+1,x+1-n+k}_{x}$.

```
Function findParent(tary, tid, child, diff)
  Input : tary, tid, child, diff                ▷ diff = n − k,
                  ▷ where n and k are the dimensions of S_{n,k}
  Output: child's parent in tary[tid]
1  len = the number of child's symbols;          ▷ len is the
2                              ▷ original dimension k of S_{n,k}.
3  dim = 0;          ▷ dim is the reduced dimension k of S_{n,k}.
4  ls = empty string;                            ▷ the last symbol
5  // reduce the dimensions of S_{n,k}
6  for dim = len; dim ≥ 2; dim = dim − 1 do
7      ls = child's dimth symbol;
8      if ls ≠ dim + diff then
9          break;         ▷ The dimension has been confirmed.
10     else if tid > diff + 1 and tid==dim + diff − 1 then
11         tid=diff+1;         ▷ Because T^{x,x−n+k}_{n−k+1} is used by
12                                     ▷ T^{x+1,x+1−n+k}_{x}.
13     else if tid==diff+1 then
14         return getFriend(substring(child, 1, dim), dim) +
           substring(child, dim + 1, len − dim);
           ▷ T_{n−k+1} rule
15 front = substring(child, 1, dim);
16 back = substring(child, dim + 1, len − dim);
17 // back is the last n − dim symbol(s) of child which
18 // is(are) identical to that of the root.
19 // back may be a empty string if the last symbol of
20 // child is not equal to that of the root initially.
21 if tid == diff + 2 then
22     temp = findTarget(front, dim);
23            ▷ Because T_{n−k+2}'s frame is the root cluster.
24 else
25     temp = findTarget(front, tid);
26 if ls == diff + 2 then
27     return temp + back;                       ▷ P2 rule.
28     // Because T_{n−k+2}'s frame is the root cluster,
29     // the target node is not used by Frame^{n,k}_{n−k+2}.
30 else if ls == diff + 1 and dim < len and
   tary[dim + diff][child] == temp + back then
31     return findTarget(front, diff + 2)+back;
32                                               ▷ P3 rule.
33       ▷ Because T^{x,x−n+k}_{n−k+1} is used by T^{x+1,x+1−n+k}_{x}.
34 else if tary[ls][child] == temp + back then
35     return findTarget(front, diff + 2)+back;
36                                               ▷ P3 rule.
37 else
38     return temp + back;                 ▷ P1 or P2 rule.
```

F. PARALLEL ALGORITHM

The parallel algorithm is presented in Algorithm 2.

Construct the $n-k$ ISTs of $S_{n-k+1,1}$ by hand. $S_{n-k+1,1}$ has $n-k$ ISTs illustrated in Figure 12.

Algorithm 2 Parallel Algorithm

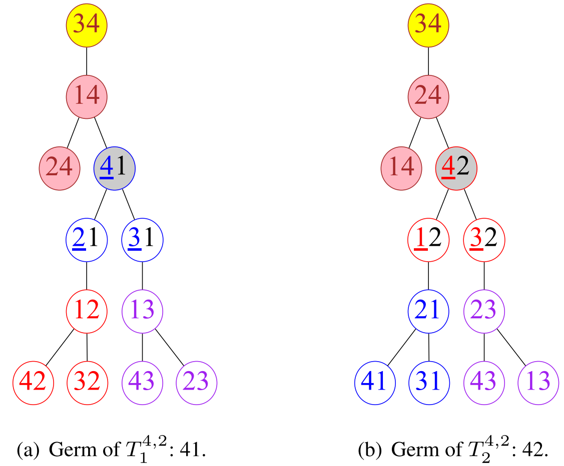
Input : n and k ▷ the dimensions of $S_{n,k}$
Output: the ISTs of $S_{n,k}$
1 Construct the $n-k$ ISTs of $S_{n-k+1,1}$ by hand;
2 **if** $n \geq n-k+2$ **then**
3 Create a *FrameGermSet*; ▷ See Algorithm 3.
4 **for** *each germ in the FrameGermSet* **do in parallel**
5 Execute buildFrameP function;
6 **for** $j = 1; j \leq n-1; j = j+1$ **do in parallel**
7 **for** *each node* v *in* $T_j^{n,k}$ **do in parallel**
8 **if** v *has no parent* **then**
9 Find and set v's parent;

Algorithm 3 Create a FrameGermSet

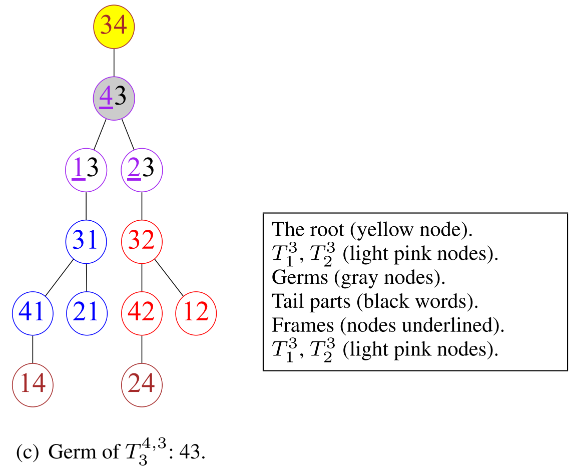
Input : n and k ▷ the dimensions of $S_{n,k}$
Output: a *FrameGermSet*
1 $children$ = getChild($root$);
2 **for** $w = 2; w \leq k; w = w+1$ **do**
3 *glist* is an array (a map) that associates values (germs) to keys (tid).
4 $temp = n-k+1$;
5 **if** $w+n-k < n$ **then**
6 $temp = w+n-k$; ▷ Because $T_{n-k+1}^{w+n-k,w}$ is
7 ▷ used by $T_{w+n-k}^{w+1+n-k,w+1}$.
8 // Put germs into $glist[tid]$ for T_1 to T_{n-k}.
9 **for** *each child* i *in children* **do**
10 $glist[i]$ = getFriend($child\ i$, w);
11 $tary[i][glist[i]] = child\ i$;
12 ▷ Set the parent of $T_i^{w+n-k,w}$'s germ.
13 $glist[temp]$ = getFriend($root$, w); ▷ Put the germ
14 ▷ of $T_{temp}^{w+n-k,w}$ into $glist[temp]$.
15 $tary[temp][glist[temp]] = root$;
16 ▷ If $w+n-k < n$, set the parent of
17 ▷ $T_{w+n-k}^{w+1+n-k,w+1}$'s germ. If $w == n$,
18 ▷ set the parent of $T_{n-k+1}^{n,k}$'s germ.
19 // Put germs into $glist[tid]$ for T_{n-k+3} to
20 // $T_{n-k+w-1}$.
21 **for** $i = 3; i <= w-1; i = i+1$ **do**
22 $glist[i+n-k]$ = getFriend(getFriend($root$,i),w);
23 $tary[i+n-k][glist[i+n-k]]$ = getFriend($root$, i); ▷ Set the parent of germs
24 ▷ from T_{n-k+3} to $T_{n-k+w-1}$.
25 **for** *each mapping (tid, germ) in glist* **do**
26 Put ($germ$, tid, $k-w+1$) into *FrameGermSet*; ▷ $k-w+1$ is *tailN*.

G. CREATE A FrameGermSet

The germs are created from the root. In the recursive algorithm to construct the ISTs of $S_{n,k}$, we should construct the ISTs of $S_{n-k+1,1}$ to $S_{n,k}$ sequentially and iteratively. In the iteration of $S_{x,x-n+k}$, the ISTs of $S_{x,x-n+k}$ must be retained in the next iteration to construct ISTs of $S_{x+1,x+1-n+k}$ as presented in Table 2. The meanings of the symbols in Table 2 are as follows:



(a) Germ of $T_1^{4,2}$: 41.
(b) Germ of $T_2^{4,2}$: 42.



(c) Germ of $T_3^{4,3}$: 43.

FIGURE 14. Germs, tail portions, frames in $S_{4,2}$.

- Hand: the IST is constructed by hand;
- New: the IST is constructed without copying the previous ISTs;
- space: No IST exists;
- $T_j^{x-1,x-1-n+k}$: in a lattice($S_{x,x-n+k}$, T_y), nodes of $T_j^{x-1,x-1-n+k}$ are appended a symbol x and becomes part of $T_y^{x,x-n+k}$.

In the parallel algorithm, we create all germs initially, and then construct all ISTs in table 2 except for $T_1^{n-k+1,1}$, $T_2^{n-k+1,1}, \ldots, T_{n-k}^{n-k+1,1}$ (constructed by hand), and $T_{n-k+2}^{n-k+3,3}$, $T_{n-k+2}^{n-k+4,4}$, $T_{n-k+2}^{n-k+5,5} \ldots T_{n-k+2}^{n-1,k-1}$ (no buildFrame operation required) in parallel. Notably, $n-k+1$th IST in $S_{x,x-n+k}$ must be part of the last IST of $S_{x+1,x+1-n+k}$. Namely

TABLE 2. The previous ISTs used in each dimension of $S_{n,k}$.

	T_1	T_2	$\dots$	T_{n-k}	T_{n-k+1}	T_{n-k+2}	T_{n-k+3}	T_{n-k+4}	$\dots$	T_{n-1}
$S_{n-k+1,1}$	Hand	Hand	$\dots$	Hand					$\dots$	
$S_{n-k+2,2}$	$T_1^{n-k+1,1}$	$T_2^{n-k+1,1}$	$\dots$	$T_{n-k}^{n-k+1,1}$	New				$\dots$	
$S_{n-k+3,3}$	$T_1^{n-k+2,2}$	$T_2^{n-k+2,2}$	$\dots$	$T_{n-k}^{n-k+2,2}$	New	$T_{n-k+1}^{n-k+2,2}$			$\dots$	
$S_{n-k+4,4}$	$T_1^{n-k+3,3}$	$T_2^{n-k+3,3}$	$\dots$	$T_{n-k}^{n-k+3,3}$	New	$T_{n-k+2}^{n-k+3,3}$	$T_{n-k+1}^{n-k+3,3}$		$\dots$	
$S_{n-k+5,5}$	$T_1^{n-k+4,4}$	$T_2^{n-k+4,4}$	$\dots$	$T_{n-k}^{n-k+4,4}$	New	$T_{n-k+2}^{n-k+4,4}$	$T_{n-k+3}^{n-k+4,4}$	$T_{n-k+1}^{n-k+4,4}$	$\dots$	
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	New	$\dots$	$\dots$	$\dots$	$\dots$	
$S_{n,k}$	$T_1^{n-1,k-1}$	$T_2^{n-1,k-1}$	$\dots$	$T_{n-k}^{n-1,k-1}$	New	$T_{n-k+2}^{n-1,k-1}$	$T_{n-k+3}^{n-1,k-1}$	$T_{n-k+4}^{n-1,k-1}$	$\dots$	$T_{n-k+1}^{n-1,k-1}$

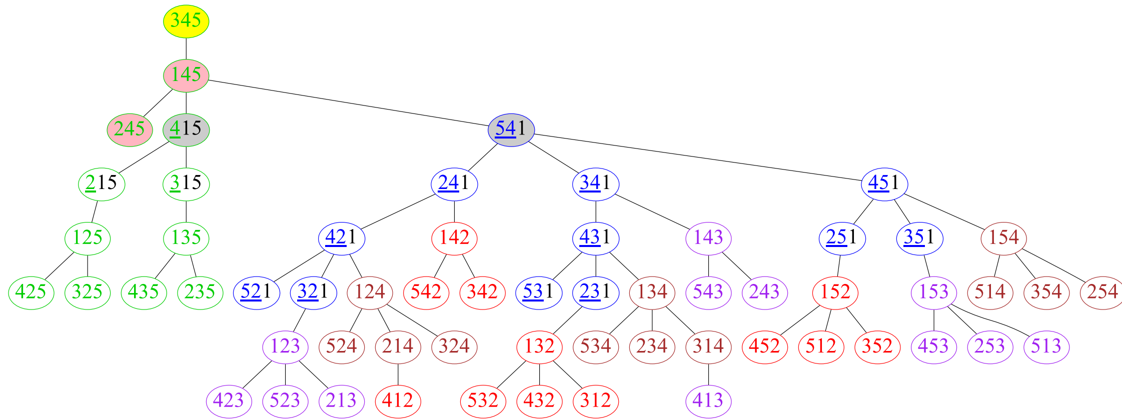


FIGURE 15. Germs (gray nodes): 415 (*tailN* 2) and 541 (*tailN* 1); tail portions (black words); frames (underlined) of $T_1^{5,3}$.

$T_{n-k+1}^{x,x-n+k}$ is used by $T_x^{x+1,x+1-n+k}$. The process is presented in Algorithm 3.

A *FrameGermSet* stores elements whose data structure as below:

- a germ;
- *tid* to indicate which IST uses the germ to create a frame;
- a tail number denoted by *tailN* to indicate the number of last symbols of a node identical to that of the germ, namely the tail portion. If $tailN \geq 2$, the last $tailN - 1$ symbol(s) of the tail portion is(are) identical of that of the root, and the dimension term of $S_{n,k}$ changes from (n, k) to $(n - tailN + 1, k - tailN + 1)$.

In $S_{4,2}$, to establish frames, we can create three germs: 41, 42, and 43. The tail number of each is 1. They are represented as gray nodes in Figure 14. In $S_{5,3}$ has four ISTs and six germs: 415 (*tailN* is 2) and 541 (*tailN* is 1) of $T_1^{5,3}$, 425 (*tailN* is 2) and 542 (*tailN* is 1) of $T_2^{5,3}$, 543 (*tailN* is 1) of $T_3^{5,3}$, and 435 (*tailN* is 2) of T_4^5 as gray nodes presented in Figures 15, 16, 17, and 18, respectively. Tree $n-k+2$ does not execute any buildFrame function in $S_{n-k+3,3}$ to $S_{n,k}$, but $T_{n-k+1}^{n-k+2,2}$ is used by $T_{n-k+2}^{n-k+3,3}$. The tail portion is represented as black symbols behind nodes.

H. PATH COMPOSITION

Finally, the outcome paths from the root to other nodes contain the frame portions and the leaf portions, as illustrated in Figure 19. The tree frames have germs, except in the case of T_{n-k+2}.

Because T_{n-k+2} done not execute any buildFrame function, it appends the last symbol of the root to its previous tree as its frame. For instance, the green nodes in $T_4^{5,3}$ (Figure 11) come from $T_3^{4,2}$ (Figure 13(c)), and the symbol 5 appended to these nodes to form the frame of $T_4^{5,3}$.

IV. PROOFS

Lemma 3: In $S_{n,k}$, each node belongs to a complete graph of $n - k + 1$ nodes.

Proof: In $S_{n,k}$, each node has $n - k$ children. Those $n-k+1$ nodes are adjacent to each other and form a complete graph. □

Lemma 4: In the frame portion, if one node has one child, it must have the other $n - k - 1$ children.

Proof: The MBFS traversal is applied in the buildFrame function; hence each parent will have $n - k$ children in the frame. □

Lemma 5: In $S_{n,k}$, each node supplies an incoming edge for each tree.

Proof: We know that $S_{n,k}$ is connected and $n - 1$ ISTs exist. Assume that some node in $S_{n,k}$ supplies its two incoming edges to some tree, then this node must be traversed twice by the tree. The situation is not reasonable, and will not occur. This notion is represented in Figure 20, where different colors represent different trees. □

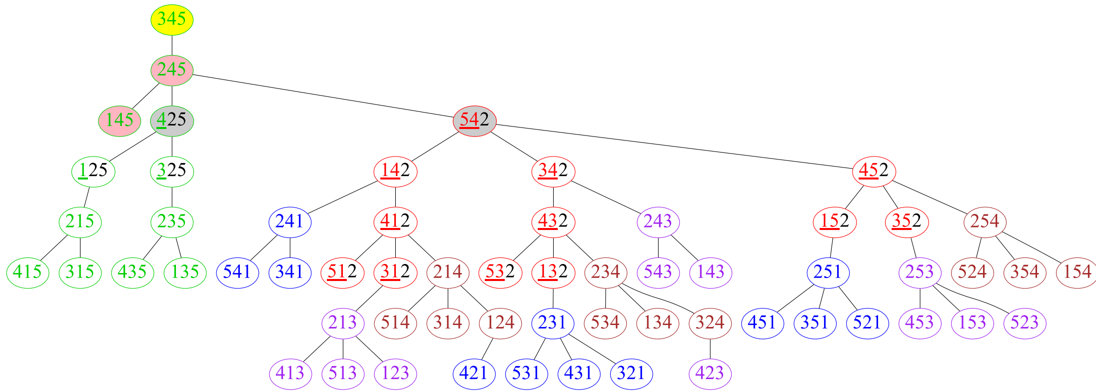


FIGURE 16. Germs (gray nodes): 425 (*tailN* 2) and 542 (*tailN* 1); tail portions (black words); frames (underlined) of $T_2^{5,3}$.

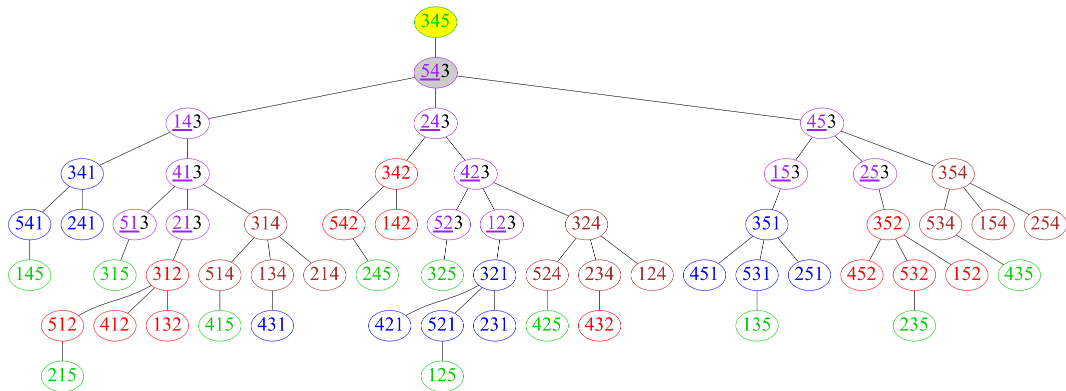


FIGURE 17. Germ (gray node): 543 (*tailN* 1); tail portions (black words); frames (underlined) of $T_3^{5,3}$.

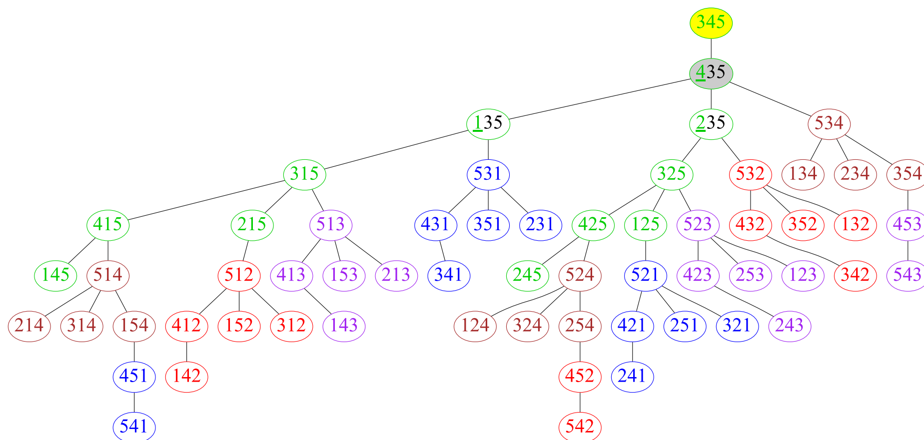


FIGURE 18. Germ (gray node): 435 (*tailN* 2); tail portions (black words); frames (underlined) of $T_4^{5,3}$.

Lemma 6: Each tree can traverse all nodes in cluster $n-k+2$ in at most two steps.

Proof: We know that T_{n-k+2} does not execute any buildFrame function. Thus, all edges in cluster $n-k+2$ are not used. In cluster $n-k+2$, each node is linked directly by some tree. If some node in cluster $n-k+2$ is not directly linked by a tree, it can link its children and friends to reach the tree. Let $X = n-k+2$. Consider one node cdX in cluster X. Tree

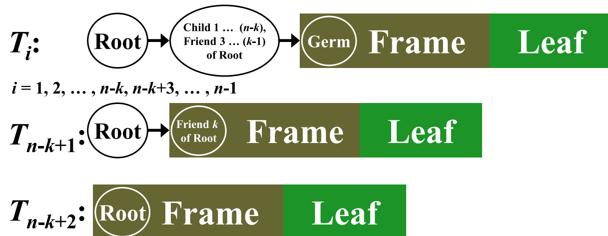


FIGURE 19. Path composition.

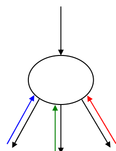


FIGURE 20. Incoming and outgoing edges.

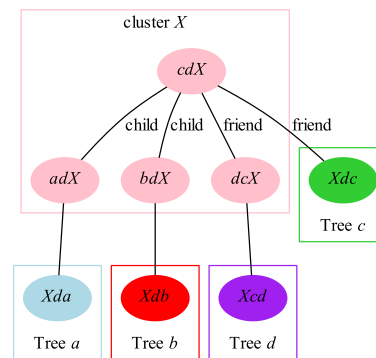


FIGURE 21. Connections of cluster $n-k+2$ (cluster X).

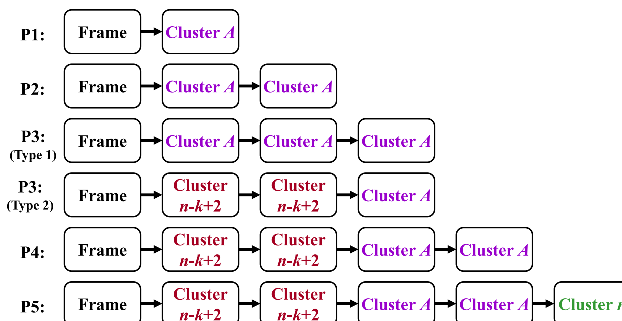


FIGURE 22. Five leaf patterns.

c links this node directly, and trees a, b, and d approach it in two steps, as illustrated in Figure 21. □

Lemma 7: The possibilities are limited to only five patterns: P1, P2, P3, P4, and P5 in the leaf portions of all ISTs in $S_{n,k}$, as illustrated in Figure 22.

Proof:

Pattern P1:

Suppose the pattern of node v in T_x is P1. v is a leaf node traversed by T_x in the first execution of buildLeaf function, and v's first symbol is x.

Pattern P2:

Suppose the pattern of node v in T_x is P2. v is a leaf node traversed by T_x in the second execution of buildLeaf function. v can approach T_x by its children or friends whose first symbol is x.

Pattern P3:

Suppose that node v's last symbol is A, and v wants a middle node m to approach the frame of Tree x through m. However, the edge from m to v is used in the frame of Tree A. We know that Tree $n-k+2$ does not conduct any buildFrame function and the last symbol of its frame nodes is the same as the last symbol n of the root. Thus, node v takes advantage of the symbol $n-k+2$ to find a node w according to Lemma 5. w's first symbol is $n-k+2$. Finally, node w can use its symbol x to approach the frame of Tree x.

P1, P2, and P3(type 1 and type 2) patterns are illustrated in the following examples.

In $S_{6,4}$, the ISTs have five paths from the root to node 5431 as follows:

3456→1456→6451→3451→5431 (frame)
3456→2456→6452→3452→5432→1432→2431→5431 (P2)
3456→6453→1453→4153→5143→1543→3541→4531 →5431(P3 type1)
3456→4356→1356→3156→5136→4136→1436→6431 →5431(P2)
3456→5436→6435→1435→5431 (P1)

Node 5431's last symbol is 1; hence, node 5431 is $T_1^{6,4}$'s frame node. In $T_1^{6,4}$, node 3451 is node 5431's parent, and node 3451 use 3 to visit node 5431. Therefore, $T_3^{6,4}$ is not able to use 3 to visit node 5431. Finally, node 5431 uses 4 to reach the frame of $T_3^{6,4}$.

In $S_{6,4}$, the ISTs have five paths from the root to node 4351 as follows:

3456→1456→6451→3451→4351 (frame)
3456→2456→6452→3452→4352→1352→2351→4351 (P2)
3456→6453→1453→4153→3154→1354→4351 (P3 type2)
3456→4356→1356→6351→4351 (P2)
3456→5436→6435→1435→4135→3145→1345→5341 →4351 (P2)

Node 4351's last symbol is 1; hence, node 4351 is $T_1^{6,4}$'s frame node. In $T_1^{6,4}$, node 3451 is node 4351's parent, and node 3451 must use 3 to visit node 4351; Therefore, $T_3^{6,4}$ is not able to use 3 to visit node 4351. Finally, node 4351 uses 4 to reach the frame of $T_3^{6,4}$.

Pattern P4:

The concept of P4 is similar to that of P3. Suppose that node v's last symbol is A, and v wants a middle node m_1 to approach the frame of Tree x through m_1. However, the edge from m_1 to v is used in the frame of Tree A. Thus, node v takes

advantage of the symbol $n-k+2$ to find a node w according to Lemma 5. w's first symbol is $n-k+2$. Then, node w wants a middle node m_2 to approach the frame of Tree x through m_2. However, the edge from m_2 to w is still used in the frame of Tree A. Thus, node w must take advantage of the symbol $n-k+2$ to find a node z according to Lemma 5. Notably, w's first symbol is $n-k+2$; therefore z's last symbol is $n-k+2$. Node z must use its symbol x to approach the frame of Tree x. Node z only passes one node in cluster $n-k+2$ to reach the frame of Tree x according to Lemma 6.

The following example demonstrates P4 pattern. There are five paths from the root to node 6421 in $S_{6,4}$.

3456→1456→6451→2451→5421→6421 (frame)
3456→2456→6452→5462→1462→2461→6421 (P2)
3456→6453→2453→5423→1423→3421→6421 (P2)
3456→4356→1356→3156→2156→5126→4126→1426 →6421 (P1)
3456→5436→6435→4635→2635→3625→4625→5624 →1624→4621→6421(P4)

Node 4621's parent in $T_1^{6,4}$ is node 5621, and node 5621's first symbol is still 5. Hence, node 4621 in $T_5^{6,4}$ should use 4 to approach the frame of $T_5^{6,4}$ as follows.

3456→1456→6451→4651→2651→5621→4621(frame)
3456→2456→6452→4652→5642→1642→2641→ 4621(P2)
3456→6453→4653→2653→5623→1623→3621 →4621(P2)
3456→4356→1356→3156→2156→5126→4126→1426 →6421→4621 (P2)
3456→5436→6435→4635→2635→3625→4625→5624 →1624→4621 (P3 type 2)

Pattern P5:

P5 only occurs in Tree $n-k+1$ for some nodes of the root cluster. The parents of those nodes belong to P4. For instance, in $S_{6,4}$, the paths from the root to node 5426 are as follows.

3456→1456→5416→2416→1426→5426
3456→2456→5426
3456→6453→4653→2653→5623→4623→3624→5624 →4625→6425→5426(P5)
3456→4356→2356→3256→4256→5246→2546→4526 →5426
3456→5436→2436→3426→5426 □

Lemma 8: No common node exists in both the frame portion of Tree A and the leaf portions of other trees in the paths from the root to node v in all trees.

Proof: The leaf portion has five patterns: P1, P2, P3, P4, and P5 according to Lemma 7. For P1 and P2, suppose that Tree x is going to traverse node v.

P1:

v's first symbol is equal to x and Tree x can traverse v by a friend operation in a step. Thus, no common node exists in both Tree A's frame portion and Tree x's leaf portion.

P2:

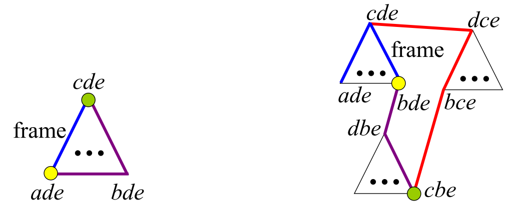
Tree x traverses node v through a middle node w in cluster A. Because edge $w \rightarrow v$ is unused in Tree A's frame and Tree x can connect node w by a friend operation in a step, there is no common node exists between Tree A's frame portion and Tree x's leaf portion.

P3:

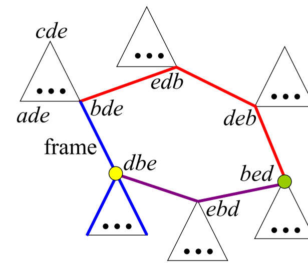
Suppose that nodes are composed of $\{a, b, c, d, e, \ldots A\}$, and we omit the rear portion. We exhaust the possibilities with the following three cases. Triangles represent complete graphs.

Case 1: a complete graph

Suppose that node ade requires a node of which the first symbol c approaches Tree c's frame, as illustrated in Figure 23(a). Because we have applied the MBFS traversal, no node such as node cde does not possess all children according to Lemma 4. Thus, it cannot occur that a common node appears in node cde.



(a) Case 1: A complete graph. (b) Case 2: Child frame.



(c) Case 3: Friend frame.

FIGURE 23. P3 does not occur in frames.

Case 2: child frame

Node bde in cluster A requires the symbol c to approach to Tree c's frame, but c is used by node cde, as illustrated in Figure 23(b). Hence, node bde uses the symbol b to walk to node dbe. Then, node dbe walks to node cbe, and node bde walks back to Tree c's frame at the end. If a common node appeared in node cbe in Tree A's frame, the path would exist: $cbe \rightarrow bce \rightarrow dce \rightarrow cde$ (marked red lines). We apply the MBFS traversal in buildFrame functions. If the function has traversed node cbe, it would use the purple lines in lieu of the red lines and blue line to traverse node bde in the MBFS traversal. Thus, a common node does not exist due to the shortest path rule.

Case 3: friend frame

Node dbe in cluster A requires the symbol b to approach Tree b's frame, but b is used by node bde, as illustrated in Figure 23(c). Hence, node dbe uses d to walk to node ebd. Then, node ebd walks to node bed, and node dbe walks back to Tree b's frame finally. If a common node

appeared in node *bed* in Tree *A*'s frame, the path would exist: *bed*→*deb*→*edb*→*bde* (marked red lines). We apply the MBFS traversal in buildFrame functions. If the function has traversed node *bed*, it would use the purple lines in lieu of the red lines and blue line to traverse node *dbe* in the MBFS traversal. Thus, a common node does not exist due to the shortest path rule.

P4:

As depicted in Figure 22, P2 pattern does not appear in the frames, and T_{n-k+2} does not execute any buildFrame function. Hence, no common node would appear in the frame.

P5:

It belongs to some nodes of the root cluster in Tree $n-k+1$. Thus, no common node exists in the frame. □

Lemma 9: The nodes of the root cluster must be traversed ultimately in T_{n-k+1}.

Proof: The root cluster in all ISTs except for T_{n-k+1} originate from the previous trees. Because the root cluster's incoming edges are not used and T_{n-k+1} has no previous tree, the root cluster must be traversed ultimately in T_{n-k+1}. For instance, the five paths of $T_1^{6,4}$ to $T_5^{6,4}$ from the root to node 5426 as follows:

3456→1456→5416→2416→1426→5426

3456→2456→5426

3456→6453→4653→2653→5623→4623→3624→5624
→4625→6425→5426

3456→4356→2356→3256→4256→5246→2546→4526
→5426

3456→5436→2436→3426→5426

Node 5426 in $T_1^{6,4}$, $T_2^{6,4}$, $T_4^{6,4}$, $and T_5^{6,4}$ originate from the previous trees in $S_{5,3}$. In $T_3^{6,4}$, node 5426 must be traversed finally. □

Theorem 1: The recursive algorithm can construct $n-1$ ISTs in $S_{n,k}$.

Proof: We prove the theorem by mathematical induction. If we know $S_{x,y}$, then we know $S_{x+1,y+1}$. When $y = 1$ and $x = n-k+1$, a complete graph of $n-k+1$ nodes holds. There are $n-1$ ISTs as illustrated in Figure 12.

Suppose that $y = k$ and $x = n$ holds. When $y = k+1$ and $x = n+1$, $S_{n+1,k+1}$ is made up of $n+1$ instances of $S_{n,k}$. We partition nodes into clusters by the last symbol of each node. First, we copy $T_1^{n,k}$ to $T_1^{n+1,k+1}$, $T_2^{n,k}$ to $T_2^{n+1,k+1}$, ..., $T_{n-k}^{n,k}$ to $T_{n-k}^{n+1,k+1}$, $T_{n-k+1}^{n,k}$ to $T_n^{n+1,k+1}$, $T_{n-k+2}^{n,k}$ to $T_{n-k+2}^{n+1,k+1}$, $T_{n-k+3}^{n,k}$ to $T_{n-k+3}^{n+1,k+1}$, ..., $T_{n-1}^{n,k}$ to $T_{n-1}^{n+1,k+1}$. Second, we find germs as explained in Table 3.

TABLE 3. Germs in $S_{n+1,k+1}$.

Tree id	Germ
1	$(n+1)(n-k+2)(n-k+3)...n1$
2	$(n+1)(n-k+2)(n-k+3)...n2$
...	...
$n-k$	$(n+1)(n-k+2)(n-k+3)...n(n-k)$
$n-k+1$	$(n+1)(n-k+2)(n-k+3)...n(n-k+1)$
$n-k+3$	$(n+1)(n-k+2)(n-k+1)(n-k+4)(n-k+5)...n(n-k+3)$
$n-k+4$	$(n+1)(n-k+2)(n-k+3)(n-k+1)(n-k+5)...n(n-k+4)$
...	...
n	$(n+1)(n-k+2)(n-k+3)...(n-1)(n-k+1)n$

Third, we use the buildFrame function to create frames. Fourth, we use buildLeaf functions to create leaf portions. For cluster $n+1$, all trees in $S_{n,k}$ are independent and $T_{n-k+1}^{n+1,k+1}$ traverses cluster $n+1$ through other clusters; hence, all paths of all trees from the root to cluster $n+1$ should be internally node-disjoint. For other clusters, each path is made up of frame and leaf portions. The frame portions are distinct and the leaf portions of all paths can be classified into to one leaf pattern of P1, P2, P3, P4, and P5, which have no common node with the frame portion in accordance with Lemma 8. Thus, the n ISTs in $S_{n+1,k+1}$ are independent, and the recursive algorithm can construct $n-1$ ISTs in $S_{n,k}$ correctly by the induction hypothesis. □

Theorem 2: $n-1$ independent spanning trees in $S_{n,k}$ constructed by the parallel algorithm are independent.

Proof: The ISTs of $S_{n-k+1,1}$ are constructed by hand, as presented in Figure 12. The buildFrame function is an instance of the buildFrameP function with *tailN* set to 1. We can use *tailN* to fix the last *tailN* symbol(s) to reduce the dimensions of $S_{n,k}$ and create all frames of ISTs from $S_{n-k+2,2}$ to $S_{n,k}$ simultaneously, whereas in the recursive algorithm the dimensions increase one in each iteration, and ISTs constructed in this iteration must be copied to ISTs next iteration. According to Algorithm 3, we create a *FrameGermSet*, and the germs of $S_{n,k}$ are identical to that of $S_{n-k+2,2}$ to $S_{n,k}$ in the recursive algorithm. Therefore, the frames created by both algorithms are the same. The findParent function produces five leaf patterns displayed in Figure 22 as the buildLeaf function does. Thus, the ISTs of $S_{n,k}$ constructed by the parallel algorithm and the recursive algorithm are the same. According to Theorem 1, the parallel algorithm can construct $n-1$ ISTs in $S_{n,k}$ accurately. □

V. ANALYSIS AND COMPARISON

We programmed in PHP and created illustrations in Graphviz. The test cases are presented in Table 4. We tested whether all paths in all trees were internally vertex-disjoint. The results proved that all paths in all trees were internally vertex-disjoint. Both algorithms are correct.

TABLE 4. The cases tested.

$n-k$	Cases
1	$S_{2,1}, S_{3,2}, S_{4,3}, S_{5,4}, S_{6,5}, S_{7,6}, S_{8,7}, S_{9,8}$
2	$S_{3,1}, S_{4,2}, S_{5,3}, S_{6,4}, S_{7,5}, S_{8,6}, S_{9,7}, S_{10,8}$
3	$S_{4,1}, S_{5,2}, S_{6,3}, S_{7,4}, S_{8,5}, S_{9,6}, S_{10,7}$
4	$S_{5,1}, S_{6,2}, S_{7,3}, S_{8,4}, S_{9,5}, S_{10,6}$
5	$S_{6,1}, S_{7,2}, S_{8,3}, S_{9,4}, S_{10,5}$
6	$S_{7,1}, S_{8,2}, S_{9,3}, S_{10,4}, S_{11,5}, S_{12,6}$
7	$S_{8,1}, S_{9,2}, S_{10,3}, S_{11,4}, S_{12,5}$
8	$S_{9,1}, S_{10,2}, S_{11,3}, S_{12,4}, S_{13,5}$
9	$S_{10,1}, S_{11,2}, S_{12,3}, S_{13,4}$
10	$S_{11,1}, S_{12,2}, S_{13,3}, S_{14,4}, S_{15,5}$

Because all nodes and all directed edges are traversed once, the time complexity of $S_{n,k}$ is the summation of the numbers of nodes and directed edges from $S_{n-k+1,1}$ to $S_{n,k}$. The number of nodes in $S_{n,k}$ is $\frac{n!}{(n-k)!}$ and the number of

directed edges in $S_{n,k}$ is $\frac{n!}{(n-k)!} \times (n-1)$. We can compute the summation:
$\sum_{i=1}^{k}(\frac{(n-k+i)!}{(n-k)!} + \frac{(n-k+i)!}{(n-k)!} \times (n-k+i-1)) = \frac{(n+1)!-(n-k+1)!}{(n-k)!}$, and the time complexity is $O(\frac{(n+1)!-(n-k+1)!}{(n-k)!}) = O(\frac{(n+1)!}{(n-k)!}) = O(\frac{(n+1)\times n!}{(n-k)!}) = O(n \times \frac{n!}{(n-k)!})$.

However, the time complexity of $S_{n,k}$ in the parallel algorithm is the summation of the numbers of nodes and directed edges of $S_{n,k}$. Thus it is $O(\frac{n!}{(n-k)!} + \frac{n!}{(n-k)!} \times (n-1)) = O(n \times \frac{n!}{(n-k)!})$, and it can decline to $O(\frac{n!}{(n-k)!})$ if the system has $n-1$ processors computing in parallel.

The functions of the recursive algorithm are buildFrame and buildLeaf, and both of them are BFS-based. In the parallel algorithm, the buildFrameP function is BFS-based, but the findParent function is rule-based. The concept of the recursive algorithm is simpler than that of the parallel algorithm. The two algorithms are compared in Table 5.

TABLE 5. The comparison of recursive and parallel algorithms.

Algorithm	Main method	Time Complexity	The number of CPUs needed	Concept
Recursive	BFS-based	$O(n \times \frac{n!}{(n-k)!})$	1	Simple
Parallel	BFS-based and Rule-based	$O(\frac{n!}{(n-k)!})$	$n-1$	Complex

VI. CONCLUSION

In this paper, we proposed one recursive algorithm and one parallel algorithm for constructing ISTs in (*n*,*k*)-star graphs. The recursive algorithm is a top-down approach, and the parent of a node in an IST is not determined by any rule. The parallel algorithm combines top-down and bottom-up approaches. In this paper, correctness, proofs, and time complexity analyses were presented for both algorithms. We also described PHP implementations of both algorithms. The test results for different fifty-nine cases are in Table 4. The results prove that all trees of all cases in Table 4 are ISTs for both algorithms. We conclude that our algorithms are not only correct but also efficient, and furthermore, the parallel algorithm is more efficient than the recursive algorithm. We hope that our contribution can aid in novel IST research.

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] J.-H. Chang and J. Kim, "Ring embedding in faulty (n,k)-star graphs," in *Proc. 8th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, 2001, pp. 99–106.

[3] Y.-H. Chang, J.-S. Yang, J.-M. Chang, and Y.-L. Wang, "A fast parallel algorithm for constructing independent spanning trees on parity cubes," *Appl. Math. Comput.*, vol. 268, pp. 489–495, Oct. 2015.

[4] 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.

[5] J.-M. Chang, T.-J. Yang, and J.-S. Yang, "A parallel algorithm for constructing independent spanning trees in twisted cubes," *Discrete Appl. Math.*, vol. 219, pp. 74–82, Mar. 2017.

[6] Y. S. Chen and K. S. Tai, "A near-optimal broadcasting in (*n*,*k*)-star graphs," in *Proc. Int. Conf. Softw. Eng. Appl. Netw. Parallel Distrib. Comput. (ACIS)*, 2000, pp. 217–224.

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

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

[9] 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.

[10] C. Wei-Kuo and C. Rong-Jaye, "The (n, k)-star graph: A generalized star graph," *Inf. Process. Lett.*, vol. 56, no. 5, pp. 259–264, Dec. 1995.

[11] W. K. Chiang and R. J. Chen, "Topological properties of the (*n*,*k*)-star graph," *Int. J. Found. Comput. Sci.*, vol. 9, no. 2, pp. 235–248, 1998.

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

[13] 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.

[14] H.-C. Hsu, Y.-L. Hsieh, J. J. M. Tan, and L.-H. Hsu, "Fault Hamiltonicity and fault Hamiltonian connectivity of the (n,k)-star graphs," *Networks*, vol. 42, no. 4, pp. 189–201, Dec. 2003.

[15] H. C. Hsu, C. K. Lin, H. M. Huang, and L. H. Hsu, "The spanning connectivity of the (*n*,*k*)-star graphs," *Int. J. Found. Comput. Sci.*, vol. 17, pp. 415–434, Apr. 2006.

[16] J.-F. Huang, S.-S. Kao, S.-Y. Hsieh, and R. Klasing, "Top-down construction of independent spanning trees in alternating group networks," *IEEE Access*, vol. 8, pp. 112333–112347, 2020.

[17] 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.

[18] 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.

[19] 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)*. Cham, Switzerland: Springer, vol. 10976, 2018, pp. 1–13.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Y. Wang, J. X. Fan, G. D. Zhou, and X. H. Jia, "Independent spanning trees on twisted cubes," *J. Parallel Distrib. Comput.*, vol. 72, pp. 58–69, Jan. 2012.

[25] 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.

[26] 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.

[27] 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.

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

JIE-FU HUANG received the B.S. degree from the Information Management Department, National Taiwan University, Taipei, Taiwan, in June 2003, the M.S. degree from the Information Management Institute, National Cheng Kung University, Tainan, Taiwan, in June 2005, and the Ph.D. degree from the Computer Science and Information Engineering Department, National Cheng Kung University, Tainan, Taiwan, in July 2020. His current research interests include the design and analysis of algorithms and graph theory.

EDDIE CHENG received the Ph.D. degree in combinatorics and optimization from the University of Waterloo, Canada, in 1995. He is currently a Distinguished Professor of Mathematics with the Department of Mathematics and Statistics, Oakland University. He has served as the Chair of the Department from 2010 to 2013 and as the Acting Chair from January 2016 to April 2016. He has directed over 30 high school students in research projects. Some of them have advanced to semifinals and beyond in national competitions such as Siemens Competitions and the Intel Science Talent Search. For his contributions in teaching and mentoring, he received the 2009 Presidents Council State Universities of Michigan Distinguished Professor of the Year Award and the 2018 University of Waterloo Faculty of Mathematics Alumni Achievement Medal. He is currently on the editorial team of a number of journals, including *Networks* (Wiley), *International Journal of Machine Learning and Cybernetics* (Springer), *Discrete Applied Mathematics* (Elsevier), *Journal of Interconnection Networks* (World Scientific), *International Journal of Computer Mathematics: Computer Systems Theory* (Taylor & Francis), and *International Journal of Parallel, Emergent and Distributed Systems* (Taylor & Francis).

SUN-YUAN HSIEH (Senior Member, IEEE) received the Ph.D. degree in computer science from National Taiwan University, Taipei, Taiwan, in June 1998. He then 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. In February 2002, he joined the Department of Computer Science and Information Engineering, National Cheng Kung University, where he is currently a distinguished Professor and the Dean of Research. Recently, he joined 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.