Search of Involution in General Graphs

Keng-Shih Lu

July 31, 2019

1 Introduction

In this report, we introduce two algorithms for searching valid involutions in a general graph. The purpose of these algorithms is to identify symmetry properties of a given graph that lead to fast graph Fourier transform (GFT) implementations discussed in the companion paper [1]. Essentially, finding all valid involutions is a combinatorial problem, where a brute force search requires an exponential complexity. However, given the knowledge of the graph structure and associated properties (e.g., when the graph is a tree), we can apply useful strategies to prune the brute force algorithm, or even design an algorithm in polynomial time.

In what follows, we will present two methods. The first method takes advantage of the degree list, and prune the exhaustive search based on the degrees of the nodes. The intuition behind it is that two nodes that are paired by an involution ϕ in a ϕ-symmetric graph must have the same degree. This provides a criterion for pruning involutions that are not valid during the search. The second approach is designed particularly for trees, which are special cases where $\mathcal{O}(n \log n)$ complexity (or $\mathcal{O}(n)$ complexity, when a $\mathcal{O}(n)$ sort is used) can be achieved. In this method, we consider a property that involutions of a tree graph can be characterized by pair(s) of identical tree branches whose roots are either common or adjacency. This property enables us to search involutions by comparing out-going branches at each tree node, leading to a linear complexity.

1.1 Notations and Definitions

We denote a graph as $\mathcal{G}$ with vertex set $\mathcal{V}$ of n nodes, edge set $\mathcal{E}$, and weighted adjacency matrix $\mathbf{W}$. We allow self-loops in the graph, and define the self-loop matrix $\mathbf{S}$ as a diagonal matrix whose diagonal elements are the self-loop weights $w_{1,1}$, $w_{2,2,}$, $\ldots$, $w_{n,n}$. The degree of node i is defined

Table 1: Telephone numbers $T(n)$ with n from 1 to 12

n	1	2	3	4	5	6	7	8	9	10	11	12
$T(n)$	1	2	4	10	26	76	232	764	2620	9496	35696	140152

as $d_i = \sum_{(i,j)\in\mathcal{E}} w_{i,j}$. The degree matrix is a diagonal matrix whose diagonal elements are $d_1, d_2, \ldots, d_n$. The Laplacian matrix is defined as $\mathbf{L} = \mathbf{D} - \mathbf{W} + \mathbf{S}$.

Definition 1 ([2])**.** *A permutation on a finite set $\mathcal{V}$ is called an involution if it is its own inverse, i.e., $\phi(\phi(i)) = i$ for all $i \in \mathcal{V}$.*

In what follows, if $\mathcal{V} = \{1, \ldots, n\}$, we denote an involution $\phi : \mathcal{V} \to \mathcal{V}$ as

$$\phi = (\phi(1), \phi(2), \ldots, \phi(n)).$$

For example, $\phi = (1, 4, 3, 2)$ is an involution with $\phi(1) = 1$, $\phi(2) = 4$, $\phi(3) = 3$, and $\phi(4) = 2$. By definition, an involution uniquely characterizes a pairing function. That is, if $i \in \mathcal{V}$ is mapped to $j \in \mathcal{V}$ ($\phi(i) = j$), then j is mapped to i ($\phi(j) = \phi(\phi(i)) = i$). Unpaired elements ($k \in \mathcal{V}$ with $\phi(k) = k$) are allowed in this definition.

Definition 2. *Let ϕ be an involution defined on the vertex set $\mathcal{V}$ of a graph $\mathcal{G}$. We say $\mathcal{G}$ is ϕ-symmetric if $w_{i,j} = w_{\phi(i),\phi(j)}$ for all $i \in \mathcal{V}$, $j \in \mathcal{V}$.*

In what follows, we say ϕ is a valid involution for graph $\mathcal{G}$ if $\mathcal{G}$ is ϕ-symmetric.

1.2 Background

The number of all involutions, also known as the telephone number, on a set of n elements can be expressed as a sum

$$T(n) = \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{n!}{2^k (n-2k)! k!} \sim \left(\frac{n}{e}\right)^{n/2} \frac{e^{\sqrt{n}}}{(4e)^{1/4}},$$

which asymptotically grows faster than polynomials in n [2]. Thus, an exhaustive search of valid involutions among $T(n)$ of possible candidates, as described as in Algorithm 1, has an exponential (or higher) complexity. The telephone numbers with n from 1 to 12 are shown in Table 1.2.

Here, we would like to design efficient algorithms for searching the valid involutions and discuss conditions of the graph for having such algorithms. In fact, from the graph topology we can obtain useful information, such as local characteristics of nodes or graph sparsity, for pruning invalid involutions. We propose two methods as in Sections 2 and 3.

Algorithm 1 Exhaustive search of graph symmetry

procedure EXHAUSTIVE($\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$)
  $\Pi \leftarrow \varnothing$
  $\mathcal{T} \leftarrow$ set of all involutions on $\mathcal{V}$
  while $\phi \in \mathcal{T}$ **do** ▷ Loop over all involutions
    if $w_{i,j} = w_{\phi(i),\phi(j)}$ for all $i, j \in \mathcal{V}$ **then**
      $\Pi \leftarrow \Pi \cup \{\phi\}$
  return Π ▷ Set of all valid involutions

2 Pruning based on Degree Lists

From Definition 2, we obtain a necessary condition for nodes i and $\phi(i)$ being paired: nodes i and $\phi(i)$ have the same *weighted degree* (sum of weights on edges adjacent to a node). This is given by

$$d_i = \sum_{(i,j)\in\mathcal{E}} w_{i,j} = \sum_{(i,j)\in\mathcal{E}} w_{\phi(i),\phi(j)} = \sum_{(\phi(i),\phi(j))\in\mathcal{E}} w_{\phi(i),\phi(j)} = d_{\phi(i)}. \tag{1}$$

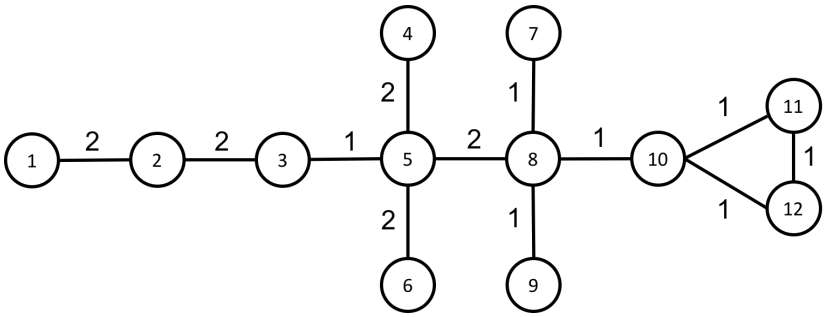
In addition, we can also consider the *unweighted degree* (number of neighbors of a node). We denote this number of neighbors for node i as f_i. Similar to (1), we have

$$f_i = \sum_{(i,j)\in\mathcal{E}} \mathbb{1}(w_{i,j}) = \sum_{(i,j)\in\mathcal{E}} \mathbb{1}(w_{\phi(i),\phi(j)}) = \sum_{(\phi(i),\phi(j))\in\mathcal{E}} \mathbb{1}(w_{\phi(i),\phi(j)}) = f_{\phi(i)}, \tag{2}$$

where $\mathbb{1}$ is the indicator function with $\mathbb{1}(w) = 1$ if $w \neq 0$ and $\mathbb{1}(w) = 0$ otherwise.

Based on these two conditions (1), (2), we obtain a useful pruning rule for the exhaustive search: *only search those involutions in which all pairs of nodes have the same degrees and numbers of neighbors.* To apply this criterion, we first partition the vertex set $\mathcal{V}$ into subsets $\{\mathcal{V}_{d,f}\}$, each of which contains nodes that share the common weighted and unweighted degrees (d and f). Then, for each vertex subset $\mathcal{V}_{d,f}$, we list all $T(|\mathcal{V}_{d,f}|)$ possible involutions on it. Finally, we search all combinations of the listed involutions across all vertex subsets. Note that, the number of such involutions is $\Pi_{d,f} T(|\mathcal{V}_{d,f}|)$, which is typically significantly smaller than $T(n)$ for graphs that are not regular (node degrees are not all equal).

An example is shown in Figure 1. We note that the graph nodes have weighted degrees $d \in \{1, 2, 3, 4, 5, 7\}$ and unweighted degrees $f \in \{1, 2, 3, 4\}$. As in Figure 1(c), there are 7 combinations of (d, f), so the vertex set is partitioned into 7 subsets with sizes 2, 3, 2, 1, 1, 1, 1, and 1. For the node in each class with size 1, all valid involutions must map it to itself, which



(a) Graph

d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}
2	4	3	2	7	2	1	5	1	3	2	2

f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}
1	2	2	1	4	1	1	4	1	3	2	2

(b) Degree lists

Involutions on $\mathcal{V}_{1,1} = \{7, 9\}$
$(d = 1, f = 1)$

$\phi(7)$	$\phi(9)$
7	9
9	7

Involutions on $\mathcal{V}_{2,1} = \{1, 4, 6\}$
$(d = 2, f = 1)$

$\phi(1)$	$\phi(4)$	$\phi(6)$
1	4	6
1	6	4
4	1	6
6	4	1

Involutions on $\mathcal{V}_{2,2} = \{11, 12\}$
$(d = 2, f = 2)$

$\phi(11)$	$\phi(12)$
11	12
12	11

Involutions on $\mathcal{V}_{3,2} = \{3\}$
$(d = 3, f = 2)$

$\phi(3)$
3

Involutions on $\mathcal{V}_{3,3} = \{10\}$
$(d = 3, f = 3)$

$\phi(10)$
10

Involutions on $\mathcal{V}_{4,2} = \{2\}$
$(d = 4, f = 2)$

$\phi(2)$
2

Involutions on $\mathcal{V}_{5,4} = \{8\}$
$(d = 5, f = 4)$

$\phi(8)$
8

Involutions on $\mathcal{V}_{7,4} = \{5\}$
$(d = 7, f = 4)$

$\phi(5)$
5

(c) Tables of involutions

Figure 1: An example of pruning based on degree lists. (a) An example graph, (b) its weighted and unweighted degree lists, and (d) its vertex partitions and involutions on them.

reduces the dimension of the search. On the other hand, those classes $\mathcal{V}_{1,1}$, $\mathcal{V}_{2,1}$, and $\mathcal{V}_{2,2}$ have 2, 3, and 2 elements, respectively, so there are $T(2) = 2$, $T(3) = 4$, and $T(2) = 2$ candidates of involutions to be searched. As a result, the number of total involutions to be searched is reduced from $T(12) = 140152$ to $T(2)^2T(3) = 2 \times 2 \times 4 = 16$.

Algorithm 2 Truncated search of graph symmetry based on degree lists

1: **procedure** TRUNCATED($\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$)
2: Compute the weighted degree list $\mathbf{d} = (d_1, \ldots, d_n)^\top$ with $d_i = \sum_{j=1}^n w_{i,j}$
3: Compute the unweighted degree list $\mathbf{f} = (f_1, \ldots, f_n)^\top$ with $f_i = \sum_{j=1}^n \mathbb{1}(w_{i,j})$
4: $\mathcal{S} \leftarrow$ GetTruncatedList($\mathbf{d}, \mathbf{f}$)
5: $\Pi \leftarrow \varnothing$
6: **while** $\phi \in \mathcal{S}$ **do** ▷ Loop over all involutions that are not pruned
7: **if** $w_{i,j} = w_{\phi(i),\phi(j)}$ for all $i, j \in \mathcal{V}$ **then**
8: $\Pi \leftarrow \Pi \cup \{\phi\}$
9: **return** Π ▷ Set of all valid involutions
10: **procedure** GETTRUNCATEDLIST($\mathbf{d}, \mathbf{f}$)
11: $\mathcal{V}_{d,f} \leftarrow \varnothing$ for all d and f
12: **for** $i = 1, \ldots, n$ **do**
13: $\mathcal{V}_{d_i,f_i} \leftarrow \mathcal{V}_{d_i,f_i} \cup \{i\}$
14: **for** $\mathcal{V}_{d,f} \neq \varnothing$ **do** ▷ Loop over all $\mathcal{V}_{d,f}$
15: $\mathcal{S}_{d,f} \leftarrow$set of all involutions on $\mathcal{V}_{d,f}$
16: $\mathcal{S} \leftarrow \bigotimes_{d,f} \mathcal{S}_{d,f}$ ▷ Direct product of all involution sets
17: **return** $\mathcal{S}$ ▷ Set of all valid involutions after pruning

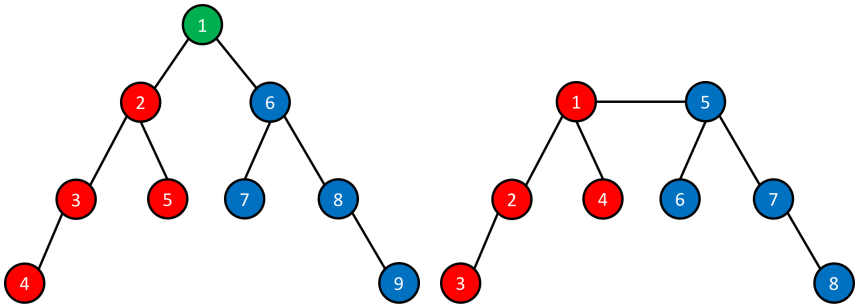
We show the procedures based on this pruning criterion in Algorithm 2. The code for MATLAB implementation can be found in our github repository[1].

2.1 Complexity Analysis

Let the number of nodes be n and the number of edges be m. We note that, for a given involution ϕ, checking whether it is a valid involution (lines 5 and 6 in Algorithm 1) requires $\mathcal{O}(m)$ time. This means that Algorithm 1 takes an overall complexity of $\mathcal{O}(mT(n))$.

To analyze the complexity of Algorithm 2, we break it into several parts:

[1] `http://github.com/kslu/fastgft/`



(a) (b)

Figure 2: Symmetry of tree characterized by identical branches. (a) Two branches with a common root. (b) Two branches with adjacent roots. Their associated involutions are $\phi_a = (1, 6, 8, 9, 7, 2, 5, 3, 4)$ and $\phi_b = (5, 7, 8, 6, 1, 4, 2, 3)$.

- Retrieval of the degree lists (lines 2 and 3). This can be done by accumulating all edge weights, so the complexity is $\mathcal{O}(m)$ of each of $\mathbf{d}$ and $\mathbf{f}$.
- Obtaining the truncated list of involutions (lines 10-17). The algorithm requires scanning through all nodes (lines 12-13), visiting all possible involutions for each $\mathcal{V}_{d,f}$ (lines 14-15), and applying the direct product of all sets of involutions (line 16). Let the number of partitions be q and the sizes of partitions be k_1, k_2, ..., k_q. Then, these three procedures take $\mathcal{O}(n)$, $\mathcal{O}(\sum_{i=1}^{q} T(k_i))$, and $\mathcal{O}(\Pi_{i=1}^{q} T(k_i))$, respectively.
- Searching over the truncated set of involutions (lines 6-8). This takes $\mathcal{O}(m\Pi_{i=1}^{q} T(k_i))$.

Combining all the complexities above, we obtain an overall complexity of $\mathcal{O}(m\Pi_{i=1}^{q} T(k_i))$. This reduces the complexity of Algorithm 1 by a factor of $T(n)/\Pi_{i=1}^{q} T(k_i)$, where $n = \sum_{i=1}^{q} k_i$.

3 Searching of Identical Tree Branches

In this section, a polynomial time ($\mathcal{O}(n \log n)$, or $\mathcal{O}(n)$ with optimization) algorithm is provided for the search of involution on trees. This algorithm arises from the fact that a symmetry in a tree can be uniquely characterized by identical branches of the tree whose roots are common (as in Figure 2(a))

or adjacent (as in Figure 2(b)). In what follows, we refer to such branches as *adjacent identical branches* (AIB). The main algorithm is based on the fact that AIBs can be identified efficiently through a bottom-up traversal in a tree.

This approach is motivated by related work in [3, 4], where the so-called *subtree repeats* can be found in linear time. The subtree repeat problem and our problem are similar, but different in two aspects. First, we do not allow the subtrees to overlap. Second, from the condition of graph symmetry, we require that the roots of the target subtrees to be adjacent. Thus, the algorithm we design can be regarded as a simplified version of the forward/non-overlapping stage in [3], and has the same time and space complexity.

First, we show that if a tree $\mathcal{G}$ is ϕ-symmetry, then this symmetry can be characterized by a pair, or several pairs of adjacent identical branches in $\mathcal{G}$:

Lemma 1. *Let ϕ be a non-trivial involution (ϕ ($\phi(i) \neq i$ for some i) and let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$ be a ϕ-symmetric tree. Then, the tree nodes can be partitioned into $\mathcal{V} = \mathcal{V}_X \cup \mathcal{V}_Y \cup \mathcal{V}_Z$ such that $\phi(i) \in \mathcal{V}_Y$ if $i \in \mathcal{V}_X$ and $\phi(j) = j$ if $j \in \mathcal{V}_Z$, and there is at most one edge $(k, l) \in \mathcal{E}$ with $k \in \mathcal{V}_X$, $l \in \mathcal{V}_Y$.*

From this lemma, we know that involution ϕ is associated to a pair of branches (sub-trees) whose vertex sets are $\mathcal{V}_X$ and $\mathcal{V}_Y$, respectively.

Proof. We define $\mathcal{V}_Z = \{i \in \mathcal{V} | \phi(i) = i\} \neq \mathcal{V}$, and discuss two cases: $\mathcal{V}_Z \neq \varnothing$ and $\mathcal{V}_Z = \varnothing$.

If $\mathcal{V}_Z$ is non-empty, since ϕ is non-trivial, there must be a (possibly non-unique) node $k_1 \in \mathcal{V}_Z$ connected to two nodes that are not in $\mathcal{V}_Z$, denoted as i_1 and $j_1 = \phi(i_1)$. First, we start with $\mathcal{V}_X = \{i_1\}$ and $\mathcal{V}_Y = \{j_1\}$. Then, we apply a depth-first search (DFS) to traverse the branch rooted by i_1 toward its other neighbors than k_1, and include those nodes being visited into $\mathcal{V}_X$. Note that, from the ϕ-symmetry, every step of the DFS corresponds to a traversal step from j_1 towards its other neighbors than k_1. In this way, we can obtain an identical branch rooted by j_1, and include all nodes in this branch into $\mathcal{V}_Y$. If there exists another node $k_2 \in \mathcal{V}_Z$ other than k_1 connected to two nodes i_2 and $j_2 = \phi(i_2)$, we apply the same traversal procedure again and append $\mathcal{V}_X$ and $\mathcal{V}_Y$ in the same way. This procedure can be repeated until all nodes are classified into $\mathcal{V}_X$, $\mathcal{V}_Y$, and $\mathcal{V}_Z$. The resulting $\mathcal{V}_X$ and $\mathcal{V}_Y$ will end up containing one or several identical branches of the original tree.

If $\mathcal{V}_Z$ is empty, then by connectivity of the tree graph, there must be an edge connecting two nodes paired by ϕ, i and $\phi(i)$. Then, we apply a DFS to traverse the branch rooted by i toward its other neighbor than $\phi(i)$, and include all visited nodes into $\mathcal{V}_X$. Similarly, the ϕ-symmetry also gives an identical branch rooted by $\phi(i)$. We include nodes in this branch into $\mathcal{V}_Y$.

The connectivity of the graph implies that all nodes are included in these two indentical branches. □

Note that the converse of this theorem is straightforward: when there are two identical branches whose roots are common or adjacent, we can identify an involution ϕ that pairs the nodes of the branches such that the graph is ϕ-symmetric.

3.1 Algorithm

Since tree symmetry can be characterized by adjacent identical branches, we can build branch descriptors for all branches in the tree, and compare branches that are joined or adjacent. Note that, to apply this method, the tree needs to be rooted, i.e., has a root so that there is a hierarchy with different levels of nodes.

The reasonable choice of root would be a node in the *center* of the tree. The center of a graph is the set of vertices v where the greatest distance $d(u, v)$ to other vertices v is minimal. In fact, if $(z_1, \ldots, z_L)$ is a longest path of the graph (whose length is called the *diameter* of the graph), then the center node(s) of this path must belong to the center of the graph. An important consequence of this property is given as follows.

Lemma 2. *A node in the center of a tree cannot be an internal node of a branch that has an adjacent identical branch.*

Proof. If otherwise, joining these two branches leads to a path whose length is greater than the diameter, yielding a contradiction. □

To find the center of a tree, we can apply breadth-first search (BFS) twice to obtain the longest path in the tree, and identify the nodes in the center of the longest path. An algorithm for obtaining the center is shown in Algorithm 3.

With Lemma 2, we design a bottom-up branch matching algorithm on a tree rooted with its center, which is summarized in Algorithm 4.

Here we provide an example with graph in Figure 3 to demonstrate the algorithm. In Table 2 we show the iterations in the order of reverse topological order (from bottom to top of the tree rooted at node 1). Note that in iteration 5, an involution associated to the two AIBs is $\phi_1 = (1, 2, 3, 5, 4, 6, 7, \ldots, 13)$ (pairing of nodes 4 and 5), while the involution found in iteration 8 is $\phi_2 = (1, \ldots, 8, 9, 11, 10, 12, 13)$ (pairing of nodes 10 and 11). In the last iteration, the involution corresponding to the largest AIBs is

$$\phi_3 = (1, 7, 9, 10, 11, 8, 2, 6, 3, 4, 5, 12, 13),$$

Algorithm 3 Finding the center of a tree

procedure TREECENTER($\mathcal{G}$) ▷ $\mathcal{G}$ is a tree
  Pick any node $v_1 \in \mathcal{V}$
  Run BFS on $\mathcal{G}$ from v_1, and obtain the furthest node v_2
  Run BFS on $\mathcal{G}$ from v_2, and obtain the furthest node v_3
  Retrieve the $v_2 - v_3$ path $(z_1 = v_2, z_2, \ldots, z_L = v_3)$ from the previous BFS solution ▷ This path is the diameter of the tree
  if $L = 2n + 1$ is odd **then**
    return $\{z_n\}$ ▷ The tree has one center
  if $L = 2n$ is even **then**
    return $\{z_n, z_{n+1}\}$ ▷ The tree has two centers

Algorithm 4 Finding valid involutions in a tree

procedure FINDINVOLUTIONINTREE($\mathcal{G}$)
  Take a node $v \in \mathcal{C}$=TreeCenter($\mathcal{G}$)
  Run BFS from v, and obtain the node list with topogolical order $(u_1, \ldots, u_n)$
  $\mathcal{I} \leftarrow \varnothing$ ▷ List of valid involutions
  for $i = n, \ldots, 1$ **do** ▷ Build branch descriptor bottom-up
    if u_i has no childs **then**
      $\mathcal{S}_i \leftarrow$‘’ ▷ Empty descriptor
    if u_i has childs **then**
      $(t_{i,1}, \ldots, t_{i,c_i}) \leftarrow$list of childs, sorted w.r.t. their descriptors.
      If any two branch descriptors are equal, append $\mathcal{I}$ with the involution characterized by the two identical branches
      $\mathcal{S}_i \leftarrow$‘$(w_{u_i,t_{i,1}}\mathcal{S}_{t_{i,1}})(w_{u_i,t_{i,2}}\mathcal{S}_{t_{i,2}})\ldots(w_{u_i,t_{i,c_i}}\mathcal{S}_{t_{i,c_i}})$’ ▷ Build descriptor by concatenating the weights and the descriptors of the sorted child branches
  if $|\mathcal{C}| = 2$ **then** ▷ The center has two nodes
    if Two branches rooted by the two center nodes are identical **then**
      Append $\mathcal{I}$ with the involution characterized by these two branches
  return $\mathcal{I}$

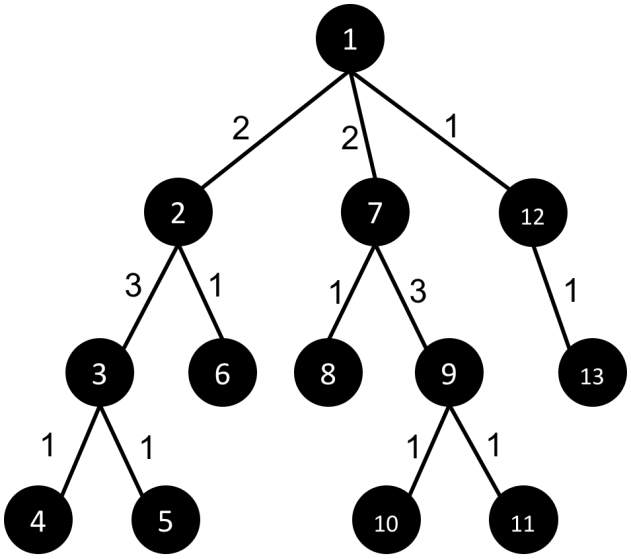


Figure 3: An example graph for demonstration of Algorithm 4. The center of this graph is $\{1\}$. Note that a topological order given by BFS from node 1 would be $(1, 2, 7, 12, 3, 6, 8, 9, 13, 4, 5, 10, 11)$.

that pairs nodes 2, 3, 4, 5, and 6 with nodes 7, 9, 10, 11, and 8, respectively. Note that, when each descriptor is built, the sorting of branch descriptors from the lower level will fix the order, so two AIBs must have identical branch descriptors.

3.2 Complexity Analysis

The algorithm consists of the following steps:

- Finding the center (line 2). This requires two BFS's and a traversal of a path. For a tree, the number of edges is $m = n - 1$, so the overall complexity of these operations on a tree is $\mathcal{O}(m + n) = \mathcal{O}(n)$.

- Finding a topological order using a BFS (line 3). The complexity of this step is $\mathcal{O}(n)$.

- Building descriptors for all nodes (line 5-11). Here, it take n iterations to loop over all tree nodes. Let p_i be the number of childs of node u_i. In iteration i, sorting the descriptors takes $\mathcal{O}(p_i \log(p_i))$, and

Table 2: Demonstration of Algorithm 4.

Iteration	Node	Branch descriptor	Comment
1	4	$\mathcal{S}_4$ =‘’	
2	5	$\mathcal{S}_5$ =‘’	
3	10	$\mathcal{S}_{10}$ =‘’	
4	11	$\mathcal{S}_{11}$ =‘’	
5	3	$\mathcal{S}_3 = (w_{3,4}\mathcal{S}_4)(w_{3,5}\mathcal{S}_5)$ =‘(1)(1)’	AIB found
6	6	$\mathcal{S}_6$ =‘’	
7	8	$\mathcal{S}_8$ =‘’	
8	9	$\mathcal{S}_9 = (w_{9,11}\mathcal{S}_{11})(w_{9,10}\mathcal{S}_{10})$ =‘(1)(1)’	AIB found
9	13	$\mathcal{S}_{13}$ =‘’	
10	2	$\mathcal{S}_2 = (w_{2,6}\mathcal{S}_6)(w_{2,3}\mathcal{S}_3)$ =‘(1)(3((1)(1)))’	
11	7	$\mathcal{S}_7 = (w_{7,8}\mathcal{S}_8)(w_{7,9}\mathcal{S}_9)$ =‘(1)(3((1)(1)))’	
12	12	$\mathcal{S}_{12}$ =‘(1)’	
13	1	$\mathcal{S}_1 = (w_{1,12}\mathcal{S}_{12})(w_{1,2}\mathcal{S}_2)(w_{1,7}\mathcal{S}_7)$ =‘(1(1))(2(1)(3(1)(1)))(2(1)(3(1)(1)))’	AIB found

concatenating them takes $\mathcal{O}(p_i)$. The overall complexity of this loop is

$$\begin{aligned}
\sum_{i=1}^{n} p_i \log(p_i) &= (n-1)\left[\sum_{i=1}^{n} \frac{p_i}{n-1}\log(p_i)\right] \\
&= (n-1)\left[\sum_{i=1}^{n} \frac{p_i}{n-1}\log\left(\frac{p_i}{n-1}\right) + \sum_{i=1}^{n} \frac{p_i}{n-1}\log(n-1)\right] \\
&= (n-1)\left[-H_p + \frac{\sum_{i=1}^{n} p_i}{n-1}\log(n-1)\right] \\
&\leq (n-1)\log(n-1) \qquad (3)
\end{aligned}$$

where the equation $\sum_{i=1}^{n} p_i = n-1$ has been applied, and $H_p \geq 0$ because it is an entropy quantity. From (3), we obtain a complexity of $\mathcal{O}(n \log n)$.

- Check complete symmetry (line 13-14). This requires a comparison of two descriptors, and takes $\mathcal{O}(n)$ time.

In fact, as mentioned in [3], sorting of descriptors can be implemented by radix sort, which requires only $\mathcal{O}(p_i)$, due to the finite range of strings to be

sorted. As a result, the overall complexity is $\mathcal{O}(n \log n)$ with an $\mathcal{O}(n \log n)$ sort, or $\mathcal{O}(n)$ with a $\mathcal{O}(n)$ sort.

4 Conclusion

In this report, we discuss two methods for searching a valid involution ϕ in a graph for graph symmetry. The first method is based on the fact that two symmetric nodes must have the same weighted and unweighted degrees. The second approach takes advantage of the sparsity of tree graphs, in which case the graph symmetry can be characterized by adjacent identical branches. We show the algorithms of both methods as well as examples for illustration. The complexities of both methods are analyzed. The MATLAB implementations can be found in the companion github repository.

References

[1] K.-S. Lu and A. Ortega, "Fast graph Fourier transforms based on symmetry and bipartition," *submitted, IEEE Trans. Signal Processing*, 2019.

[2] Donald E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[3] T. Flouri, K. Kobert, S. P. Pissis, and A. Stamatakis, "An optimal algorithm for computing all subtree repeats in trees," in *Combinatorial Algorithms*, T. Lecroq and L. Mouchard, Eds., Berlin, Heidelberg, 2013, pp. 269–282, Springer Berlin Heidelberg.

[4] M. Christou, M. Crochemore, T. Flouri, C. S. Iliopoulos, J. Janoušek, B. Melichar, and S. P. Pissis, "Computing all subtree repeats in ordered trees," *Information Processing Letters*, vol. 112, no. 24, pp. 958 – 962, 2012.