Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Article

A Polynomial Time Algorithm for Graph Isomorphism and Automorphism

Sardar Anisul Haque

Department of Mathematics and Computer Science, Alcorn State University, MS, USA; sahaque@alcorn.edu

Abstract: This paper describes a polynomial time algorithm for solving graph isomorphism and automorphism. We introduce a new tree data structure called *Walk Length Tree*. We show that such tree can be both constructed and compared with another in polynomial time. We prove that graph isomorphism and automorphism can be solved in polynomial time using Walk Length Trees.

Keywords: graph isomorphism; graph automorphism; polynomial time algorithm

1. Introduction

The graph isomorphism (GI) problem is one of the most famous open problems in theoretical computer science. Two graphs are isomorphic if there exists a bijective function between their vertex sets that preserves adjacency relationship. Two Graphs, with n vertices each, have $n!$ possible one to one correspondences between the vertex sets. So, checking all such correspondences for large n is impractical. Graph automorphism is a similar problem like GI, where the input is a single graph and we have to find out all possible mappings to itself. In other words, automorphism means graph isomorphism with itself.

GI is an important tool for a number of areas including computer vision, pattern recognition, bioinformatics, mathematical chemistry and electronic design automation. As an example, chemists use graphs to model chemical compounds, where vertices and edges represent atoms and chemical bonds respectively. A new synthesized chemical compound is required to be checked with other potential molecular graphs from a database to see whether the molecular graph of the new compound is the same as one already known.

It can be shown that these problems are in NP and it is unknown whether it has polynomial time algorithm or they are in NP-complete [13]. In this paper, we propose a new polynomial time algorithm for solving graph isomorphism problem. In other words, in this paper, we prove that they are in P.

Two isomorphic graphs must have same number of vertices and edges. The degrees of the corresponding vertices in isomorphic graphs must be the same. They must also have simple circuits of same length. It is possible that these invariants are same for two graphs that are not isomorphic. There are no useful sets of invariants currently known that can be computed in polynomial time to determine whether two graphs are isomorphic [21].

The list of published algorithms that attempt to solve GI is long [13]. We are listing a few of those. The best currently accepted theoretical algorithm, which has $e^{O(\sqrt{n \log n})}$ time complexity, is due to [2]. Later, the author claimed that GI is in quasi-polynomial time, which has not been fully peer-reviewed yet [12]. There exists a number of polynomial time algorithms for some restricted graphs. For example, a polynomial time algorithm is described in [1] for graphs of bounded degree. In [22], the authors gave a linear time algorithm for tree isomorphism. Moreover, polynomial time algorithms exist for planer graphs [3], interval graphs [4], permutation graphs [5], circulant graphs

© 2021 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

[7], graphs with bounded tree width [8], graphs with bounded genus [10], and graphs with bounded eigenvalue multiplicity [11].

Apart from these theoretical achievements, a number of software packages to solve GI are available. Some of the well-known packages are namely *nauty* [6,15], *saucy* [16], *bliss* [18]. A study on their performances can be found in [13]. A study on the performance of some exact graph isomorphism algorithms for different types of graphs is reported in [19].

The main contribution of this paper is that, we propose an algorithm to detect graph isomorphism and automorphism that runs in polynomial time. We also propose a new rooted tree data structure called Walk Length Tree. We provide a polynomial time algorithm to construct this tree. We also prove that two of such trees can be compared for equivalency by showing one to one correspondences between them in polynomial time. Our algorithm first create Walk Length Trees for underlying graphs, then those are used to solve graph isomorphism and automorphism problem in polynomial time.

Remark 1 (Both our proposed algorithms and Walk Length Tree are new concepts)**.** *To the best of our knowledge, Walk Length Tree is a new data structure. The author would also like to declare that the proposed algorithms that use Walk Length Trees and run in polynomial time, to detect graph isomorphism and automorphism, are entirely new concepts.*

The rest of the paper is organized in the following way. In section 2, some notations, definitions, symbol used in this paper are discussed. The construction, properties and computational aspects of Walk Length Trees are discussed in section 3. In section 4, we provide theorems that connects Walk Length Trees with graph automorphism and isomorphism problems. Finally, our proposed algorithms are described in the same section.

2. Preliminary

This section contains some graph related definitions and theorems that are taken from [21].

A graph $G = (V, E)$ consists of V, a nonempty set of n vertices and E, a set of m edges. Each edge has either one or two vertices associated with it, called its endpoints. An edge is said to connect its endpoints.

Figure 1 and Figure 2 are examples of two undirected graphs. For both cases, $n = 8$ and $m = 12$. In addition, these two graphs are isomorphic.

A graph in which each edge connects two different vertices and where no two edges connect the same pair of vertices is called a simple graph. Graphs that have multiple edges connecting the same vertices are called multigraphs.

An edge can connect a vertex to itself. Such edges are called loops.

A directed graph consists of a nonempty set of vertices and a set of directed edges. Each directed edge is associated with an ordered pair of vertices. The directed edge associated with the ordered pair (u, v) is said to start at u and end at v. u is said to be adjacent to v and v is said to be adjacent from u. We call u and v as the initial and terminal vertex of this edge.

The degree of a vertex v in an undirected graph denoted by $deg(v)$ is the number of edges incident with it. The in-degree of vertex v in a directed graph denoted by $deg^{-}(v)$ is the number of edges with v as their terminal vertex. The out-degree of vertex v in a directed graph denoted by $deg^{+}(v)$ is the number of edges with v as their initial vertex.

Graphs that have a number assigned to each edge are called *weighted graphs*.

A walk consists of an alternating sequence of vertices and edges consecutive elements of which are incident, that begins and ends with a vertex. The length of a walk is its number of edges. A path is a walk with no repeated vertex.

An undirected graph is called *connected* if there is a path between every pair of distinct vertices of the graph. An undirected graph that is not connected is called

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

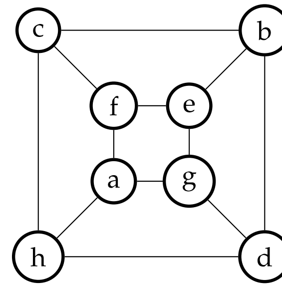

Figure 1. A Graph with 8 vertices and 12 edges.

disconnected. A *connected component* of an undirected graph G is a connected subgraph of G that is not a proper subgraph of another connected subgraph of G.

A directed graph is *strongly connected* if there is a path from v_a to v_b and from v_a to v_b, whenever v_a and v_b are vertices in the graph. The subgraphs of a directed graph G, that are strongly connected but not contained in larger strongly connected subgraphs, that is the maximal strongly connected subgraphs, are called *strong components* of G. A directed graph is weakly connected if there is a path between every two vertices in the underlying undirected graph.

The diameter d of a graph is the greatest distance between any pair of vertices. For a disconnected graph, the distance between two unreachable vertices is ∞.

Suppose that the vertices of a graph G are listed arbitrarily as $v_0, \ldots, v_{n-1}$. So, the i-th vertex in the list is v_i. The adjacency matrix of G, denoted by A_G, with respect to this listing of the vertices, is the $n \times n$ integer valued matrix, such that:

- if G is undirected, $A_G[i][j]$ is the number of edges between v_i and v_j.
- if G is directed, $A_G[i][j]$ is the number of edges from v_i to v_j.

The adjacency matrix of graphs in Figure 1 for the vertex listing $[a, b, c, d, e, f, g, h]$ is given in Eqation 1.

$$A_G = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{1}$$

For an adjacency matrix A_G and a positive integer $r > 0$, we define the power of a matrix by repeating matrix multiplication; that is $A_G^r = \underbrace{A_G \times \cdots \times A_G}_{r \text{ times}}$. We define $M(n)$, as the time complexity of matrix multiplication routine for two dense square matrices of order n. From literature, we know $M(n)$ is polynomial, with respect to n [9].

Theorem 1 (Counting walks among vertices as stated in [21])**.** *Let G be a graph with adjacency matrix A_G with respect to the ordering $v_1, \ldots, v_n$ of the vertices of the graph (with directed or undirected edges, with multiple edges and loops allowed). The number of different walks of length r from v_i to v_j , where r is a positive integer, equals the (i, j)th entry of A_G^r.*

Proof. The proof can be found in [21]. □

3. Walk Length Tree

We describe Walk Length Tree and its properties in this section. In subsection 3.1, we classify the edges E of a graph $G(V, E)$ from the perspective of any arbitrary

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

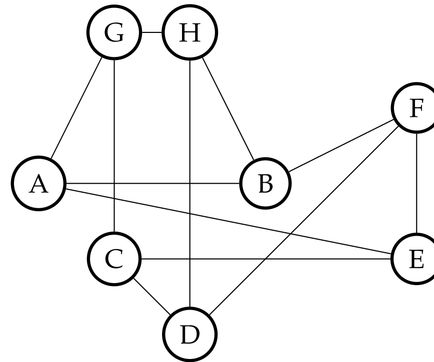

Figure 2. A graph that is isomorphic with the graph in 1.

vertex $v_i \in V$. This classification is important in Walk Length Tree construction. We then describe a suitable data structure to store edge weights and parallel edges. This data structure is useful for our proposed algorithms. Moreover, we simplify problem statement by using this data structure. We provide two other simplifications in subsection 3.2. In subsection 3.3, we describe both definition, and construction complexity of Walk Length Tree in details. We provide some very important properties of Walk Length Tree in subsection 3.4. In brief, the purpose of one Walk Length Tree is to classify vertices and edges from one reference point. In multigraphs or weighted graphs, we need more variants to distinguish those. That is why, we describe some techniques by which we classify them with each other in subsection 3.5. In subsection 3.6, we define a subgraph called *star subgraph*, by which, we describe how two Walk Length Trees can be equivalent and how two vertices can be equivalent in those Walk Length Trees in subsection 3.7 (for undirected graph) and subsection 3.8 (for directed graph). Finally, in subsection 3.9, we summarize the useful information from a Walk Length Tree those are important for graph isomorphism and automorphism problem.

3.1. Some Related Definitions, Data Structures and Algorithms

Definition 1 (Edge distance from v_i of an undirected graph $G(V, E)$)**.** *Let, s_a and s_b be the lengths of the shortest paths from $v_i \in V$ to the two endpoints of an edge $e \in E$ respectively, where $s_a \leq s_b$. We will call e is an edge of $(s_a + 1)$-*radius *from v_i. If any vertex is not reachable from v_i, then any edge connected with it is called an edge of ∞-*radius *from v_i. In Figure 1, the length of the shortest paths from a to b and c are 3 and 2 respectively. So, the edge (b, c) is an edge of 3-radius from vertex a.*

Definition 2 (Edge distance from v_i of a directed graph $G(V, E)$)**.** *Let $e \in E$ be an edge that is connecting from vertex $v_a \in V$ to vertex $v_b \in V$. The length of the shortest path from $v_i \in V$ to v_a is s_a. We will call e is an edge of $(s_a + 1)$-*radius *from v_i. If any vertex is not reachable from v_i, then any edge connected with it is called an edge of ∞-*radius *from v_i. In Figure 10, the length of the shortest paths from u_1 to u_4 is 2. So, the edge (u_4, u_3) is an edge of 3 radius from vertex u_1.*

Lemma 1 (Time complexity of distance based edge classification algorithm)**.** *Given a graph $G(V, E)$, we can classify all edges in E following Definition 1 (if G is undirected) or Definition 2 (if G is directed), for all vertices in V, in polynomial time with respect to n and m, where $|V| = n$, and $|E| = m$.*

Proof. For the purpose of edge classification, we consider G as a simple graph by (i) ignoring all loops, (ii) keeping a single edge as a representative edge of all parallel edges that have same endpoints, and (iii) keeping the other edges unchanged. We then consider the weight of every edge as 1 for both weighted and unweighted graphs. Thus, the length and cost of the shortest paths between any two vertices becomes numerically same. We know polynomial time algorithm, with respect to the number of vertices,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

to find all pair shortest paths [9]. We first compute the shortest path between all pair of vertices. We then classify any edge from the perspective of any vertex following Definition 1 and Definition 2. It takes $O(mn)$ time. Furthermore, let the length of the shortest path from vertex u to v be s. The loop edges of v are edges of $(s+1)$-radius from u. All parallel edges having same endpoints, have same radius value from a vertex $v \in V$. □

Definition 3 (Data structure for storing edge weights and parallel edges)**.** *Let v_a and v_b be two vertices of a graph $G(V, E)$. Let there exist $q \geq 0$ number of edges with or without weights*
(i) from v_a to v_b (for directed graph), or
(ii) between v_a and v_b (for undirected graph).
We store q and the edge weights (for weighted graphs) in ascending order.

Lemma 2 (Time complexity of storing edge weights)**.** *We store q and the edge weights (for weighted graphs) in ascending order as described in Definition 3 for each pair of vertices, that have edges between them, such that, given a pair of vertices, (v_a, v_b), we can get q and the weights of the edges between them in sorted order (for weighted graphs) in linear time with respect to q.*

Proof. The time complexity to create the data structure for storing edge weights as described in Definition 3 depends on the type of graph.

1. For unweighted graphs, we store q values for each pair of vertices in a matrix of $n \times n$. It costs $O(n^2)$ time to construct such matrix.
2. For weighted simple graphs, we store the weights of edges in a matrix of $n \times n$. It costs $O(n^2)$ time to construct such matrix.
3. For weighted multigraphs, we apply multi-criterion sorting described below to sort the edge weights.
   - We consider the ordering of the vertices as given in the adjacency matrix that represents the graph in the following way.
   - For directed graph, the weight of edge (v_x, v_y) appears before the weight of edge (v_c, v_d) if v_x appears before v_c or $v_x = v_c$ and v_y appears before v_d in the above ordering. Similarly, for undirected graph, the weight of edge (v_x, v_y) appears before the weight of edge (v_c, v_d) if v_x or v_y appears before both v_c and v_d or $v_x = v_c$ and v_y appears before v_d in the above ordering.
   - For directed graph, weights of parallel edges from v_x to v_y are kept together and the weights of all such parallel edges are sorted in ascending order.
   - For undirected graph, weights of parallel edges between v_x and v_y are kept together and the weights of all such parallel edges are sorted in ascending order.

   The above sorting procedure takes polynomial time with respect to m, where $|E| = m$, by any comparison based sorting algorithm. We apply quicksort that takes polynomial time with respect to m. Once all edge weights are sorted, we store both q as well as the sorted weights in a hash table, such that for any edge these information can be retrieved directly from it. So, we need polynomial time to implement such hash based data structure.

□

3.2. Simplification

To keep the description of this paper simple, we describe all definitions, theorems, corollaries, lemmas keeping a single unweighted edge as a representative edge of all parallel edges that have same endpoints. However, we can retrieve all parallel edges including their weights from the data structure described in Definition 3, if necessary.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

We describe all properties, theorems, and our proposed algorithms (in section 4) for connected undirected graphs and weakly connected directed graphs only.

Let $G(V, E)$ be a disconnected undirected graph that has N connected components, where N is a finite positive number and $|V| \geq N > 1$. Any disconnected undirected graph $G'(V', E')$, which is isomorphic with $G(V, E)$, must have N connected components. Suppose we know two polynomial time algorithms to detect graph automorphism and isomorphism for connected components. We need to call those algorithms N and $N(N+1)/2$ times to detect graph automorphism and graph isomorphism respectively. Thus, the time complexity of graph automorphism and isomorphism checking for undirected graph remains polynomial.

We define *weakly connected components* of a directed graph as the subgraphs of a directed graph G that are weakly connected but not contained in larger weakly connected subgraphs, that is, the maximal weakly connected subgraphs, are called the weakly connected components of G. Let $G(V, E)$ be a directed graph that has N weakly connected components, where N is a finite positive number and $|V| \geq N > 1$. Any directed graph $G'(V', E')$, which is isomorphic with $G(V, E)$, must have N weakly connected components. Suppose we know two polynomial time algorithms to detect graph automorphism and isomorphism for weakly connected components. We need to call those algorithms N and $N(N+1)/2$ times to detect graph automorphism and graph isomorphism respectively. Thus, the time complexity of graph automorphism and isomorphism checking for directed graph remains polynomial.

3.3. Walk Length Tree construction

Definition 4 (Walk Length Tree). *Let A_G be the adjacency matrix representation of $G(V, E)$, constructed following the vertex listing: $[v_0, \ldots, v_{n-1}]$, where $|V| = n$. $T_{v_i}(G)$ is the* Walk Length Tree *or WLT from v_i of $G(V, E)$, where $v_i \in V$. We define this tree by the way it is constructed.*

- *$T_{v_i}(G)$ is a rooted tree of height n.We call v_i as the reference vertex of $T_{v_i}(G)$.*
- *A node at the l-th level of the tree denoted as $N(l, j)$, where j is the position of the node from left, contains a tuple (W, p, Z), where $W \subseteq V$, p is positive integer, and $Z \subseteq E$.*
- *For root node $N(0, 0)$, it contains tuple $(V, 0, \{\emptyset\})$.*
- *Each node $N(l, j)$, except leaf nodes, has one or more child nodes $N(l+1, j'), \ldots, N(l+1, j'+k')$, where $j', k' \geq 0$. Each of the vertex $v_k \in W$ of $N(l, j)$, is assigned to exactly one of its child node's W set.*
- *According to Theorem 1, the number of l-length walks from reference vertex (v_i) to a vertex v_j is $A^l_G[i][j]$. For a node $N(l, j)$, the number of l-length walks from reference vertex (v_i) to any vertex $v_k \in W$ is same. This value is denoted as p. We explain below how we distribute the vertices in W set among its child nodes' W sets and maintain their p values.*
  1. *The p value of the root node is 0 by definition.*
  2. *the vertices in W of a node $N(l, j)$ (except leaf nodes), is partitioned based on the number of $(l+1)$-length walks from v_i, such that all vertices in the same part have same number of $(l+1)$-length walks from v_i. Any two vertices taken from any two different parts have different number of $(l+1)$-length walks from v_i.*
  3. *Each of these parts is assigned to one of its child node's W set.*
  4. *The order of the child nodes from left to right is in ascending order of their p values.*
  5. *These rules above apply repeatedly for each node $N(l, j), l < n$, in the WLT.*
- *The last item in this tuple is $Z \subseteq E$, which is described below.*
  1. *for multigraph, a single edge as a representative edge from all parallel edges that have same endpoints, will be considered in the Z set.*
  2. *an edge is assigned in the Z set of exactly one node of the WLT.*
  3. *if e is an edge of ∞-*radius *from v_i then it will not be an element of any Z sets of that WLT.*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

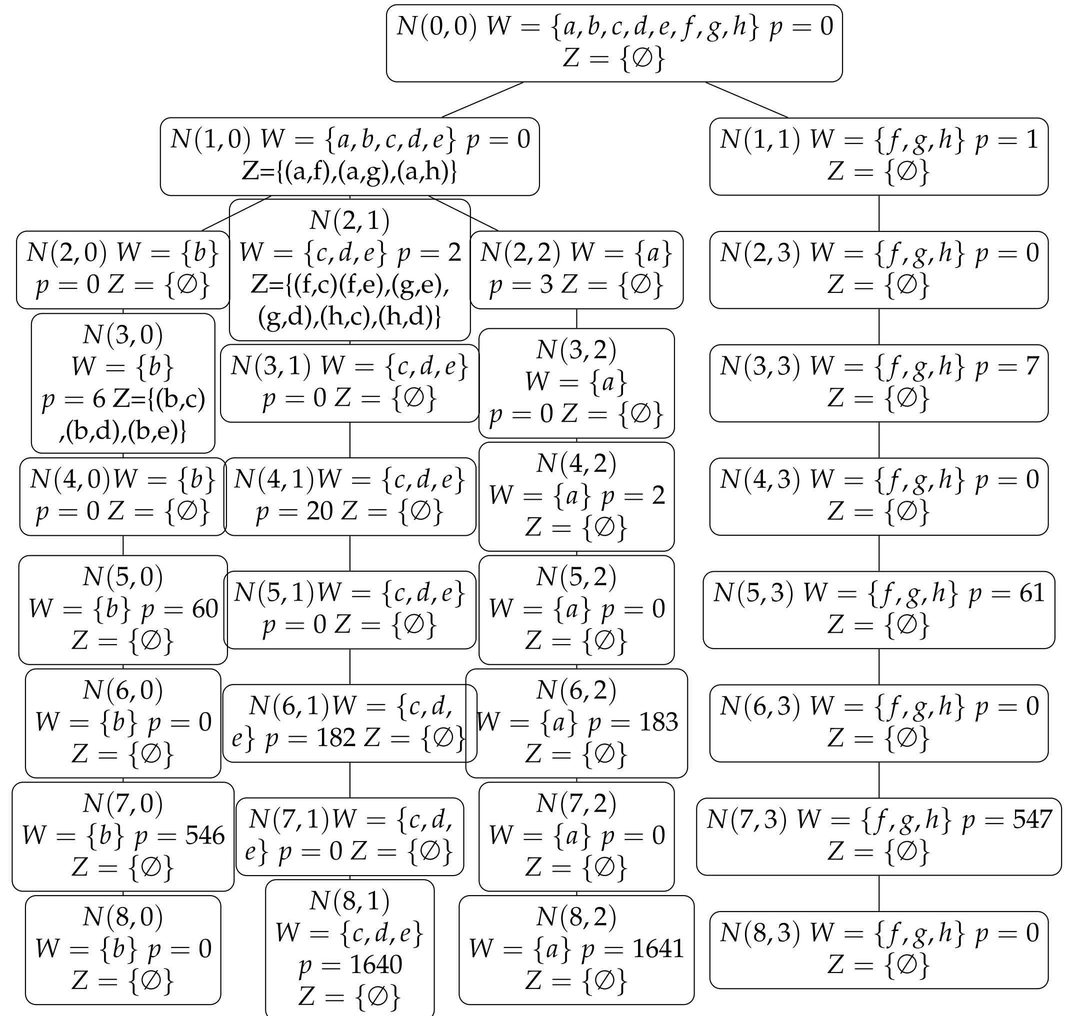

Figure 3. The Walk Length Tree of the graph in Figure 1 from vertex a.

4. *if (v_a, v_b) is an edge of l-radius edge from v_i, it is assigned to the Z set of a node $N(r, j)$, where $r = l$ and j is chosen in the following way.*
5. *for a directed graph, this edge is an element of the Z set of the same node $N(l, j)$, whose W set contains v_b.*
6. *for an undirected graph,*
   (case 1) Let both v_a, and v_b be in the W set of a node $N(l, j)$, it is assigned to the Z set of $N(l, j)$.
   (case 2) If v_a and v_b be in the W set of $N(l, j)$ and $N(l, j_1)$ respectively, where $j_1 > j$, it is assigned to the Z set of $N(l, j)$.

The WLT of the graph in Figure 1 from vertex a is given in Figure 3. It can be verified by raising the adjacency matrix (in equation 1) to power of $2, \ldots, 8$. We insert the node information $N(l, j)$ with each node of this WLT.

The WLT of the graph in Figure 2 from vertex B is given in Figure 4.

A graph have n WLTs. In Theorem 2, we describe the time complexity of creating all n WLTs of a given graph.

Theorem 2 (Time complexity of constructing all Walk Length Trees of G)**.** *Given the adjacency matrix representation A_G of a graph $G(V, E)$, we can construct n WLTs with reference to all vertices $v_0, \ldots, v_{n-1} \in V$ of G, as given in Definition 4, in polynomial time with respect to n and m, where $n = |V|$ and $m = |E|$.*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

$W = \{A, B, C, D, E, F, G, H\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{B, C, D, E, G\}$ $p = 0$ Z={(B,A),(B,H),(B,F)}

$W = \{A, H, F\}$ $p = 1$ $Z = \{\emptyset\}$

$W = \{C\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{D, E, G\}$ $p = 2$ Z={(A,E) (A,G),(H,D), (H,G),(F,D),(F,E)}

$W = \{B\}$ $p = 3$ $Z = \{\emptyset\}$

$W = \{A, H, F\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{C\}$ $p = 6$ Z={(C,D), (C,E),(C,G)}

$W = \{D, E, G\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{B\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{A, H, F\}$ $p = 7$ $Z = \{\emptyset\}$

$W = \{C\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{D, E, G\}$ $p = 20$ $Z = \{\emptyset\}$

$W = \{B\}$ $p = 2$ $Z = \{\emptyset\}$

$W = \{A, H, F\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{C\}$ $p = 60$ $Z = \{\emptyset\}$

$W = \{D, E, G\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{B\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{A, H, F\}$ $p = 61$ $Z = \{\emptyset\}$

$W = \{C\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{D, E, G\}$ $p = 182$ $Z = \{\emptyset\}$

$W = \{B\}$ $p = 183$ $Z = \{\emptyset\}$

$W = \{A, H, F\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{C\}$ $p = 546$ $Z = \{\emptyset\}$

$W = \{D, E, G\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{B\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{A, H, F\}$ $p = 547$ $Z = \{\emptyset\}$

$W = \{C\}$ $p = 0$ $Z = \{\emptyset\}$

$W = \{D, E, G\}$ $p = 1640$ $Z = \{\emptyset\}$

$W = \{B\}$ $p = 1641$ $Z = \{\emptyset\}$

$W = \{A, H, F\}$ $p = 0$ $Z = \{\emptyset\}$

Figure 4. The Walk Length Tree of the graph in Figure 2 from vertex B.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Proof. We need to compute A_G^r, where $r = 2, \dots, n$, which requires polynomial time with respect to n [9], to know p values for all nodes of each WLTs.

For a WLT, we construct the W sets of all nodes of a level together. Suppose, all nodes of level l of a WLT are created. Now, we are going to create the nodes of level $l+1$. To do this, we need to partition all vertices in such a way that the vertices of each part have same number of $l+1$ lengths walks from the reference vertex and are from the same node at level l. Now, we assign each part to the W set of the appropriate node at level $l+1$. We implement a multi-criterion based comparator function and use it in a comparison based sorting algorithm to do this partition. This sorting is done in polynomial time with respect to n.

For a WLT, we have n levels. So in total, we have n^2 levels over n such trees. Thus the complexity of constructing trees, without Z sets, from A_G^r, for $r = 2, \dots, n$ is polynomial with respect to n.

We store all W sets in a hash table, such that we can get a particular W set by calling a hash function with $N(l, j)$. We also need to know the list of nodes at each level, where a vertex is assigned to their W set. We store it by a list of n integers. So a $n \times n$ integer matrix can store this information.

We apply edge classification method discussed in Lemma 1 to know the edge distance from the reference vertex of that WLT. It requires polynomial time with respect to n and m. Let (v_a, v_b) be an edge of l-radius from the reference vertex. From the definition of WLT, we know how this edge will be added to the Z set of a single node of a WLT at level l. This is done in constant time provided that given the two endpoints v_a and v_b, we know the nodes in level l whose W sets contain these two vertices.

For one WLT, we need $O(m)$ time to assign all edges to corresponding Z sets of all nodes.Thus, in total, we need $O(mn)$ time to assign all edges to corresponding Z sets to all nodes of all Walk Length Trees.

So, the time complexity of constructing all Walk Length Trees of G is polynomial. □

Definition 5 (Shallow equivalent between two nodes of Walk Length Trees)**.** *Suppose, we have two nodes $N(l_1, j_1)$ and $N(l_2, j_2)$ from two different WLTs. We say $N(l_1, j_1)$ and $N(l_2, j_2)$ are shallow equivalent if and only if all of the following assertions are true:*

- $l_1 = l_2$,
- $j_1 = j_2$,
- *the p value of both nodes are equal.*
- *the cardinality of W sets of both nodes are equal.*
- *the cardinality of Z sets of both nodes are equal.*
- *both nodes have equal number of child nodes.*
- *if the W set of $N(l_1, j_1)$ contains the reference vertex of its WLT, the W set of $N(l_2, j_2)$ must contain the reference vertex of its WLT and vice versa.*

Definition 6 (Shallow equivalent between two Walk Length Trees)**.** *Two WLTs, $T_{v_i}(G)$ and $T_{v_{i'}}(G')$ are called shallow equivalent, denoted as $T_{v_i}(G) \stackrel{S}{\equiv} T_{v_{i'}}(G')$, if and only if for each node in $T_{v_i}(G)$ there must be a shallow equivalent node in $T_{v_{i'}}(G')$ and vice versa. Here, G and G' may or may not be the same graph.*

For each node in the WLT of Figure 1 has a shallow equivalent node in the WLT of Figure 2. Thus, these are shallow equivalent WLTs.

From Definition 5 and Definition 6, we can conclude that the number of nodes at any level of two shallow equivalent WLTs are same.

Theorem 3 (Time complexity to check Walk Length Tree shallow equivalency)**.** *We can check the shallow equivalency between two WLTs, as described in Definition 5, and Definition 6, in $O(n^2)$ time.*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

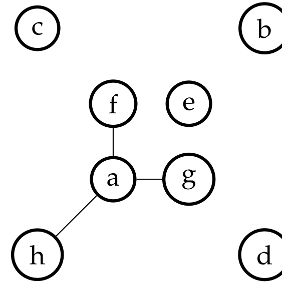

Figure 5. Leveled graph $G_1(V, E_1)$ of graph in Figure 1 considering WLT in 3.

Proof. Suppose we want to check whether two WLTs $T_{v_i}(G)$ and $T_{v_{i'}}(G')$ are shallow equivalent or not. We visit both trees in breadth-first search order together and check all conditions listed in Definition 5 for each pair of nodes. Let $N(i, j)$ and $N(i', j')$ be the k-th nodes in breadth-first search order.

1. check if $i = i'$ and $j = j'$: this is done in $O(1)$ time.
2. check the equality of their p values: this is done in $O(1)$ time.
3. check the equality of the cardinalities of their W sets: this is done in $O(1)$ time.
4. check the equality of the cardinalities of their Z sets: this is done in $O(1)$ time.
5. check if they have same number of child nodes: this is done in $O(1)$ time.
6. both nodes must either contain the reference vertex or not contain the reference vertex. This constraint can be checked in $O(n)$ time as the list of nodes at each level that contain a particular vertex is stored in a matrix as described in Theorem 2. This check is done one time for a pair of WLTs.

Given two nodes, each from each of the WLTs, we need $O(1)$ time to check their shallow equivalency. The number of nodes in a Walk Length Tree is bounded by $O(n^2)$. So, the complexity to check the shallow equivalency of two such WLTs can be done in $O(n^2)$ time. □

3.4. Properties of Walk Length Trees

Lemma 3 (Walk length tree is unique)**.** *For a graph $G(V, E)$ and a vertex $v \in V$, we get the same WLT $T_v(G)$, irrespective of its construction.*

Thus, we can create n different WLTs from a graph that has n vertices.

Lemma 4 (Reconstructing of a graph from a Walk Length Tree)**.** *Given a WLT $T_v(G)$ and the data structure as described in Definition 3 that stores parallel edges with weights, we can reconstruct G.*

All l-radius edges, where $n \geq l > 0$, from v can be found in the Z sets of all nodes in level l of the WLT $T_v(G)$. In this paper, we show that how WLTs are used to distinguish each vertices and edges.

Definition 7 (Leveled graph of Walk Length Tree)**.** *A leveled graph $G_l(V, E_l)$ of $T_{v_i}(G)$ is defined as a subgraph of $G(V, E)$, whose edge set $E_l \subseteq E$ contains edges of k-radius from v_i, where $k = 1, \ldots, l$.*

For example, Figure 5, Figure 6 are leveled graphs $G_1(V, E_1)$ and $G_2(V, E_2)$ respectively of graph in Figure 1 considering WLT in Figure 3. It should be noted that $G_3(V, E_3)$ of the graph in Figure 1 considering WLT in Figure 3 has all edges of the graph.

Lemma 5 (Walk Length Tree and leveled graphs)**.** *For a given graph $G(V, E)$, all leveled graphs $G_l(V, E_l)$, for $l = 1, \ldots, n$, of any particular WLT are fixed. Consequently, $G_n(V, E_n)$ of any particular WLT, is $G(V, E)$, if G is a connected undirected graph or a strongly connected*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

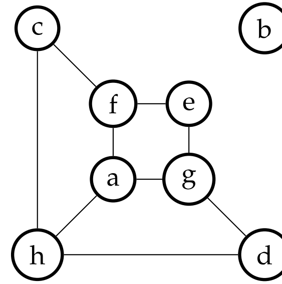

Figure 6. Leveled graph $G_2(V, E_2)$ of graph in Figure 1 considering WLT in 3.

directed graph. Otherwise, $G_n(V, E_n)$ of $T_{v_i}(G)$ is defined as a subgraph of $G(V, E)$, whose edge set $E_n \subseteq E$ contains edges of k-radius from v_i, where $k = 1, \ldots, n$.

3.5. Ordering Of Vertices and Edges

In this section, we order both vertices and edges of a graph. This ordering becomes important when we check the equivalency between two WLTs.

Definition 8 (Walk Length Tree and rank of a vertex)**.** *Given a WLT $T_{v_i}(G)$, let a vertex v be in the W set of a leaf node $N(n, j)$. We call j as the rank of v. Every vertex has a rank in a WLT.*

The rank of vertex b and c are 0 and 1 respectively in the WLT of Figure 3.

Definition 9 (Shallow equivalent Walk Length Trees and vertex shallow equivalent)**.** *Let $G(V, E)$ and $G'(V', E')$ be two graphs and $T_{v_i}(G) \overset{S}{\equiv} T_{v_{i'}}(G')$, where $v_i \in V$ and $v'_i \in V'$, a vertex $v \in V$ is called shallow equivalent to a vertex $v' \in V'$ if their ranks are same.*

The reference vertices of these two WLTs are called shallow equivalent as they are in the W set of two shallow equivalent leaf nodes.

From Definition 9, the following shallow vertex equivalencies between the Walk Length Trees in Figure 3 and Figure 4 can be found.

- vertex a and b are shallow equivalent to vertex B and C respectively.
- any vertex in $\{c, d, e\}$ is shallow equivalent to any vertex in $\{D, E, G\}$ and vice versa.
- any vertex in $\{f, g, h\}$ is shallow equivalent to any vertex in $\{A, H, F\}$ and vice versa.

Lemma 6 (Shallow equivalent vertices and their nodes)**.** *Let $G(V, E)$ and $G'(V', E')$ be two graphs and $T_{v_i}(G) \overset{S}{\equiv} T_{v_{i'}}(G')$. Let v_a and v'_a be two shallow equivalent vertices from $G(V, E)$ and $G'(V', E')$ respectively with respect to these WLTs. If v_a and v'_a are in the W sets of node $N(l, j)$ and $N(l, j')$ respectively, for $l \geq 0$ then $j = j'$.*

Proof. We prove it by contradiction. Suppose, $j < j'$. If we visit the vertices in the W sets of the leaves of any of the WLTs from left to right we will see vertices in the W set of $N(l, j)$ before those of $N(l, j')$. So the rank of any vertices in the W set of $N(l, j)$ is smaller than the rank of all vertices in the W set of $N(l, j')$. So the rank of v_a is smaller than the rank of v'_a. This is a contradiction as they are shallow equivalent. So $j = j'$ and these two nodes are shallow equivalent. □

The ordering of vertices based on the rank is not enough when we check the equivalency between two WLTs. Because two vertices having same rank might be different considering their edges. Given a graph $G(V, E)$ and a WLT $T_{v_i}(G)$, we sort the

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Algorithm 1 Comparator function to sort the vertices

Input: vertices u and v along with their ranks with respect to a WLT

1: **if** u has greater rank than that of v w.r.t the WLT respectively **then**
2: **return** $TRUE$
3: **else if** v has greater rank than that of u w.r.t the WLT respectively **then**
4: **return** $FALSE$
5: **else if** u has more loop edges than that of v **then**
6: **return** $TRUE$
7: **else if** v has more loop edges than that of v **then**
8: **return** $FALSE$
9: **else if** $deg(u) > deg(v)$,[for undirected graph] **then**
10: **return** $TRUE$
11: **else if** $deg(u) < deg(v)$,[for undirected graph] **then**
12: **return** $FALSE$
13: **else if** $deg^+(u) > deg^+(v)$,[for directed graph] **then**
14: **return** $TRUE$
15: **else if** $deg^+(u) < deg^+(v)$,[for directed graph] **then**
16: **return** $FALSE$
17: **else if** $deg^-(u) > deg^-(v)$,[for directed graph] **then**
18: **return** $TRUE$
19: **else if** $deg^-(u) < deg^-(v)$,[for directed graph] **then**
20: **return** $FALSE$
21: **else**
22: We need to sort all edges that have u or v as endpoints separately in the following way. *(for unweighted undirected graph)* sort all non-loop edges according to the rank of the other endpoints. *(for unweighted directed graph)* (i) first, keep all incoming non-loop edges in ascending order of the ranks of the other endpoints, (ii) then keep all outgoing non-loop edges in ascending order of the ranks of the other endpoints. *(for weighted undirected graph)* (i) first, keep all loop edges in ascending order of the edge weights, (ii) then keep all non-loop edges in ascending order of the edge weights. If two or more edge weights are equal then determine their ordering based on the rank of the other endpoints. *(for weighted directed graph)* (i) first, keep all loop edges in ascending order of the edge weights, (ii) then keep all non-loop and incoming edges in ascending order of the edge weights. And finally, (iii) keep all non-loop and ougoing edges in ascending order of the edge weights. If two or more edge weights are equal then determine their ordering based on the rank of the other endpoints.
23: **for** $i = 0, 1, \ldots,$ **do**
24: **if** the weight of the i-th edge of u is greater than that of v **then**
25: **return** $TRUE$
26: **else if** the weight of the i-th edge of v is greater than that of u **then**
27: **return** $FALSE$
28: **else if** the rank of the other endpoint of i-th edge of u is greater than that of v **then**
29: **return** $TRUE$
30: **else if** the rank of the other endpoint of i-th edge of v is greater than that of u **then**
31: **return** $FALSE$
32: **end if**
33: **end for**
34: **end if**
35: **return** $EQUAL$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

vertices of the graph using Algorithm 1, as a comparator function. In this sorting, we consider their edges. The time complexity to sort the vertices is described in Lemma 7.

Lemma 7 (Time complexity to sort the vertices)**.** *We can sort the vertices of a graph with respect to a WLT using Algorithm 1 in polynomial time.*

Proof. In worst case, Algorithm 1 needs to sort the edges that are connected with the input vertices. Let m' be a positive number denoting the maximum number of edges that are connected with any vertices of the graph. So, the worst case running time of this algorithm is $O(m' log(m'))$, provided we use a comparison based sorting algorithm inside of this algorithm to sort edges. As $m \geq m'$, Algorithm 1 runs in polynomial time with respect to m.

So the time complexity to sort n vertices, using Algorithm 1 as a comparator function, is polynomial with respect to n and m. □

Algorithm 2 Computing comprehensive ranks for vertices

Input: (i) a graph $G(V, E)$, (ii) a WLT $T_{v_i}(G)$, and (iii) list of vertices $(v_0, \ldots, v_{|V|-1})$ in sorted order based on the comparator function given in 1

1: $C(v_0) = 0$
2: **for** $i = 1, \ldots, |V| - 1$ **do**
3:    **if** Algorithm 1 returns $EQUAL$ for v_i, v_{i-1}, and $T_{v_i}(G)$ **then**
4:       $C(v_i) = C(v_{i-1})$
5:    **else**
6:       $C(v_i) = C(v_{i-1}) + 1$
7:    **end if**
8: **end for**

Definition 10 (Comprehensive rank of vertices)**.** *Given a graph $G(V, E)$ and a WLT $T_{v_i}(G)$, we sort the vertices as described above. We call Algorithm 2 with the sorted list of vertices and $T_{v_i}(G)$ to assign a positive integer $C(v)$ for each vertex $v \in V$ in the graph. We call $C(v)$ as the comprehensive rank of vertex v with respect to $T_{v_i}(G)$.*

The comprehensive ranks of vertices in the graph given in Figure 1 with respect to the Walk Length Tree in Figure 3 are the same as their rank.

Definition 11 (Shallow equivalent edge)**.** *Let (W, p, Z) and (W', p', Z') be the tuples of two shallow equivalent nodes of two WLTs. We call edge $(v_x, v_y) \in Z$, and $(v_{x'}, v_{y'}) \in Z'$ are shallow equivalent if*
(for directed graph) v_x, and v_y are shallow equivalent to $v_{x'}$, and $v_{y'}$ respectively.
(for undirected graph) v_x, and v_y are shallow equivalent to either $v_{x'}$, and $v_{y'}$ respectively or $v_{y'}$, and $v_{x'}$ respectively.

From Definition 11, the following shallow equivalent edges between the WLTs in Figure 3 and Figure 4 can be found.

- any edge in Z of $N(1, 0)$ in Figure 3, that is $\{(a, f), (a, g), (a, h)\}$, is shallow equivalent to any edge in Z of $N(1, 0)$ in Figure 4, that is $\{(B, A), (B, H), (B, F)\}$.
- any edge in Z of $N(2, 1)$ in Figure 3, that is $\{(f, c), (f, e), (g, e), (g, d), (h, c), (h, d)\}$, is shallow equivalent to any edge in Z of $N(2, 1)$ in Figure 4, that is $\{(A, E),(A, G),(H, D),(H, G),(F, D),(F, E)\}$.
- any edge in Z of $N(3, 0)$ in Figure 3, that is $\{(b, c), (b, d), (b, e)\}$, is shallow equivalent to any edge in Z of $N(3, 0)$ in Figure 4, that is $\{(C, E), (C, G), (C, D)\}$.

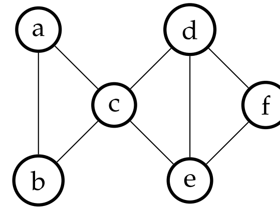

Figure 7. An undirected graph with $n = 7$ and $m = 8$.

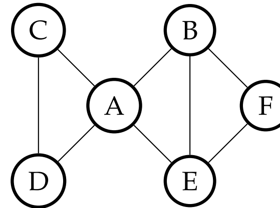

Figure 8. An undirected graph with $n = 7$ and $m = 8$.

Algorithm 3 Comparator function to order edges

Input: edges (u, v) and (u', v') and a WLT

1: (u, v) is a l-radius edge with respect to the reference vertex of its WLT
2: (u', v') is a l'-radius edge with respect to the reference vertex of its WLT
3: if the WLT is created based on a undirected graph, we assume that $C(u) \leq C(v)$ and $C(u') \leq C(v')$
4: **if** $l > l'$ **then**
5: **return** $TRUE$
6: **else if** $l < l'$ **then**
7: **return** $FALSE$
8: **else if** $C(u) > C(u')$ with respect to their respective WLTs **then**
9: **return** $TRUE$
10: **else if** $C(u') > C(u)$ with respect to their respective WLTs **then**
11: **return** $FALSE$
12: **else if** $C(v) > C(v')$ with respect to their respective WLTs **then**
13: **return** $TRUE$
14: **else if** $C(v') > C(v)$ with respect to their respective WLTs **then**
15: **return** $FALSE$
16: **else if** the q value of (u, v) is greater than that of (u', v') [as described in Definition 3] **then**
17: **return** $TRUE$
18: **else if** the q value of (u', v') is greater than that of (u, v) [as described in Definition 3] **then**
19: **return** $FALSE$
20: **end if**
21: for weighted multigraphs, the sorted edges weights are extracted from hash table as described in Definition 3 as two separate lists
22: **for** $i = 0, 1, \ldots, q - 1$ **do**
23: **if** the weight of the i-th edge of (u, v) pair is greater than that of (u', v') pair **then**
24: **return** $TRUE$
25: **else if** the weight of the i-th edge of (u', v') pair is greater than that of (u, v) pair **then**
26: **return** $FALSE$
27: **end if**
28: **end for**
29: **return** $EQUAL$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

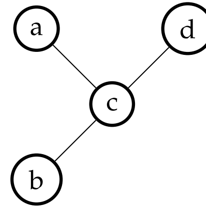

Figure 9. The star subgraph of the undirected graph in Figure 7 centered at c for an arbitrary edge set $\{(a,c),(a,b),(b,c),(c,d),(e,f)\}$.

Like vertices, we also need to sort the edges of a graph. Given a graph $G(V,E)$ and a WLT $T_{v_i}(G)$, we sort the edges of the graph using Algorithm 3, as a comparator function. This sorting is required to distinguish edges that are shallow equivalent with each other in a WLT, but they have differences considering parallel edges and edge weights. The time complexity to sort the edges is described in Lemma 8.

Lemma 8 (Time complexity to order the edges). *We can sort the edges of a graph $G(V,E)$ with respect to a WLT $T_{v_i}(G)$ using Algorithm 3 as a comparator function in polynomial time.*

Proof. Algorithm 3 runs in constant time for all kinds of graphs except weighted multigraphs. It runs in linear time with respect to the number of parallel edges in case of weighted multigraphs. So, any comparator based sorting algorithm can sort $m = |E|$ edges using this comparator function in polynomial time with respect to m and the maximum number of parallel edges having same endpoints in the graph. □

Algorithm 4 Computing comprehensive ranks for edges

Input: (i) a graph $G(V,E)$, (ii) a WLT $T_{v_i}(G)$, and (iii) list of edges $(e_0,\dots,e_{|E|-1})$ in sorted order based on the comparator function given in Algorithm 3

1: $B(e_0) = 0$
2: **for** $i = 1,\dots,|E|-1$ **do**
3: **if** Algorithm 3 returns $EQUAL$ for e_i, e_{i-1}, and $T_{v_i}(G)$ **then**
4: $B(e_i) = B(e_{i-1})$
5: **else**
6: $B(e_i) = B(e_{i-1}) + 1$
7: **end if**
8: **end for**

Definition 12 (Comprehensive rank of edges). *Given a graph $G(V,E)$ and a WLT $T_{v_i}(G)$, we sort the edges as described above. We call Algorithm 4 with the sorted list of edges and $T_{v_i}(G)$ to assign a positive integer $B(e)$ for each vertex $e \in E$ in the graph. We call $B(e)$ as the comprehensive rank of edge e with respect to $T_{v_i}(G)$.*

3.6. Star Subgraph

In this subsection, we define a subgraph called *star subgraph*. This subgraph is important in understanding how two vertices from two different WLTs can be equivalent.

Definition 13 (Star subgraph with a center vertex with respect to a given edge set). *Let $G(V,E)$ be a given graph and let $v \in V$ and $E_c \subseteq E$. We create a subgraph $G_s(V_s,E_s)$ in the following way.*

- *E_s contains all edges of E_c, whose at least one endpoint is v.*
- *for each edge $e_s \in E_s$, we add both of its endpoints in V_s.*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

We call $G_s(V_s, E_s)$ the star subgraph *of $G(V, E)$ centered at $v \in V$ for the edge set $E_c \subseteq E$. We denote $v \in V$ as the center vertex of this star subgraph.*

The star subgraph of the undirected graph in Figure 7 centered at c for an arbitrary edge set $\{(a, c), (a, b), (b, c), (c, d), (e, f)\}$ is given in Figure 9.

Lemma 9 (Time complexity in constructing a star graph)**.** *The time complexity of constructing a star subgraph of a given graph $G(V, E)$ centered at any vertex for any edge sets $E_c \subseteq E$ is polynomial with respect to $|E_c|$.*

Definition 14 (Vertex equivalent for given edge sets)**.** *Let $G(V, E)$ and $G'(V', E')$ be two graphs such that $T_{v_i}(G) \overset{S}{\equiv} T_{v_{i'}}(G')$. Let $T_{v_i}(G)$ and $T_{v_{i'}}(G')$ contain u and u' vertices respectively. Given two edge sets $E_g \subseteq E$ and $E'_g \subseteq E'$, we create two star subgraphs $G_s(V_s, E_s)$ and $G'_s(V'_s, E'_s)$ centered at $u \in V$ and $u' \in V'$ for $E_g \subseteq E$ and $E'_g \subseteq E'$ respectively.*

We call $u \in V$ and $u' \in V'$ are equivalent for the given edge sets E_g and E'_g with respect to the given shallow equivalent WLTs, if $G_s(V_s, E_s)$ and $G'_s(V'_s, E'_s)$ are isomorphic and the bijective function that shows their isomorphism must satisfy the following constraints.

- *u must map to u' and $C(u) = C(u')$.*
- *if any vertex $v \in V_s$ maps to vertex $v' \in V'_s$, then $C(v) = C(v')$.*
- *if any edge $e \in E_s$ maps to edge $e' \in E'_s$, then $B(e) = B(e')$.*

Furthermore, we call $u \in V$ and $u' \in V'$ are equivalent with respect to the given shallow equivalent WLTs $T_v(G)$ and $T_{v'}(G')$, if they are equivalent for edge sets (i) E and E', (ii) Z and Z', where they are the edge sets from any two shallow equivalent node pairs of $T_v(G)$ and $T_{v'}(G')$ respectively, and (iii) E_l and E'_l, where E_l and E'_l be the set of all l-radius edges from v and v' respectively and $n \geq l > 0$.

In the above definitions, $G(V, E)$ and $G'(V', E')$ can be a single graph, and $T_{v_i}(G)$ and $T_{v_{i'}}(G')$ can be also a single WLT.

Algorithm 5 checks the isomorphism between two star graphs as described in Definition 14. For simplicity, this algorithm is written in a general way for all kinds of graphs.

Lemma 10. *Algorithm 5 runs in polynomial time with respect to the number of vertices and edges in the input star graphs.*

The vertex a of Figure 1 is equivalent with the vertex B of Figure 2 with respect to WLTs in Figure 3 and Figure 4 respectively.

3.7. Walk Length Tree Equivalences for Undirected Graphs

Definition 15 (Two equivalent Walk Length Trees of undirected graphs)**.** *Two shallow equivalent WLTs from the same or two different undirected graphs are called equivalent if and only if we can create n pairs of vertices (v, v') such that (i) v is from one WLT and v' are from the other WLT, (ii) each vertex will participate only one of those pairs, (iii) if v is the reference vertex of one WLT, v' must be the reference vertex of the other WLT, (iv) v and v' are equivalent with respect to their respective WLTs, so, they are equivalent for the Z edge sets of any two shallow equivalent nodes. We put one more requirements to the bijective function between their edge sets, which is used to prove the isomorphism between the star graphs centered at v and v', that is if one edge maps to another edge, their other endpoints must be equivalent with respect to the given WLTs.*

The Walk Length Trees in Figure 3 and Figure 4 are equivalent. We can create the following pairs $(a, B), (b, C), (c, D), (d, E), (e, G), (f, A), (g, H), (h, F)$.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Algorithm 5 Checking if two given star graphs are isomorphic

Input: (i) star subgraph $G_s(V_s, E_s)$ and $G'_s(V'_s, E'_s)$ as discussed in Definition 14, (ii) the comprehensive ranks of all vertices and all edges of the subgraphs with respective to some WLTs, and (iii) the center vertices $u \in V$ and $u' \in V'$ of $G_s(V_s, E_s)$ and $G'_s(V'_s, E'_s)$ respectively.

1: **if** $C(u) \neq C(u')$ OR $|V_s| \neq |V'_s|$ OR $|E_s| \neq |E'_s|$ **then**
2:     **return** $FALSE$
3: **end if**
4: mark all vertices in V_s and V'_s as *unchecked* except u and u'. mark u and u' as *checked*
5: **for** each vertex $v \in V_s$ that are *unchecked* **do**
6:     **if** there exists an unchecked vertex $v' \in V'_s$ such that (i) $C(v) = C(v')$, (ii) $B(u, v) = B(u', v')$, these two edges are equivalent considering the parallel edges and weights as described in Definition 3 and (iii) $B(v, u) = B(v', u')$ these two edges are also equivalent considering the parallel edges and weights as described in Definition 3. **then**
7:
8:         mark v and v' as *checked*
9:     **else**
10:         **return** $FALSE$
11:     **end if**
12: **end for**
13: **if** all vertices in both V_s and V'_s are *checked* **then**
14:     **return** $TRUE$
15: **end if**
16: **return** $FALSE$

Lemma 11 (Time complexity to check WLT equivalency in undirected graph)**.** *We can check the equivalency between two shallow equivalent WLTs for undirected graphs as described in Definition 15 in polynomial time with respect to n and m.*

Proof. We need to partition all vertices, such that in each part, all vertices satisfy the conditions described in Definition 15. And two vertices from two different parts does not satisfy all those conditions. It takes polynomial time with respect to n and m to do so. If the reference vertices are in the same part and all parts contains even number of vertices, out of which exactly half of them are from one WLT then we can say that the given WLTs are equivalent. □

3.8. Walk Length Tree Equivalences for Directed Graphs

Definition 16 (Neighbor edge, neighbor vertex and host vertex sets of a Walk Length Tree of a directed graph)**.** *Let $G_n(V, E_n)$ be the leveled graph of a WLT $T_{v_i}(G)$ for a directed graph $G(V, E)$. The neighbor edge set E_g of $T_{v_i}(G)$ is a set of edges such that (i) $E_g \subseteq E$, (ii) $E_g \cap E_n = \emptyset$, and (iii) for an edge $e \in E_g$, it is ∞-*radius *from v_i and exactly one endpoint of e is not reachable from v_i.*

The neighbor vertex set V_g of $T_{v_i}(G)$ is a set of vertices such that (i) $V_g \subset V$, (ii) any vertex $v \in V_g$ is not reachable from v_i, and (iii) there exists at least one neighbor edge of $T_{v_i}(G)$, where v is the initial vertex.

The host vertex set V_h of $T_{v_i}(G)$ is a set of vertices such that (i) $V_h \subseteq V$, (ii) each vertex $v \in V_h$ is reachable from v_i, and (iii) there exists at least one neighbor edge of $T_{v_i}(G)$, where v is the terminal vertex.

For a neighbor edge $(u, v) \in E_g$ of $T_{v_i}(G)$, where u and v are a neighbor and a host vertex of $T_{v_i}(G)$ respectively. We call this pair of vertices as neighbor-host *pair vertices of (u, v). We call v as a host vertex of u and we call u as a neighbor vertex of v.*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

For example, the directed graph in Figure 10, the only neighbor edge of $T_{u_1}(G)$ is $\{(u_5, u_6)\}$. For this edge, u_5 and u_6 are the neighbor and host vertices respectively.

Definition 17 (Equivalence of first degree between two Walk Length Trees of directed graphs)**.** *Two shallow equivalent WLTs from the same or two different directed graphs are called equivalent of first degree if and only if we can create n pairs of vertices (v, v') such that (i) v is from one WLT and v' are from the other WLT, (ii) each vertex will participate only one of those pairs, (iii) if v is the reference vertex of one WLT, v' must be the reference vertex of the other WLT, (iv) v and v' are equivalent with respect to their respective WLTs, so, they are equivalent for the Z edge sets of any two shallow equivalent nodes. We put one more requirements to the bijective function between their edge sets, which is used to prove the isomorphism between the star graphs centered at v and v', that is if one edge maps to another edge, their other endpoints must be equivalent with respect to the given WLTs.*

If we compute all WLTs of the directed graphs in Figure 10 and Figure 11, we can see the following equivalent WLTs of first degree.

- WLTs $T_{u_1}(G), T_{u_5}(G), T_{v_1}(G')$, and $T_{v_5}(G')$ are equivalent of first degree.
- WLTs $T_{u_9}(G)$, and $T_{v_9}(G')$ are equivalent of first degree.
- WLTs $T_{u_2}(G), T_{u_6}(G), T_{u_{10}}(G), T_{v_2}(G'), T_{v_6}(G')$, and $T_{v_{10}}(G')$ are equivalent of first degree.
- WLTs $T_{u_3}(G), T_{u_7}(G), T_{u_{11}}(G), T_{v_3}(G'), T_{v_7}(G')$, and $T_{v_{11}}(G')$ are equivalent of first degree.
- WLTs $T_{u_4}(G), T_{u_8}(G), T_{u_{12}}(G), T_{v_4}(G'), T_{v_8}(G')$, and $T_{v_{12}}(G')$ are equivalent of first degree.

Lemma 12 (Time complexity to check WLT equivalency in directed graph of first degree)**.** *We can check the equivalency between two shallow equivalent WLTs for directed graphs of first degree in polynomial time considering n and m.*

Proof. See proof for Lemma 11. □

Definition 18 (Equivalence of second degree between two Walk Length Trees of directed graphs)**.** *Two equivalent of first degree Walk Length Trees $T_v(G)$ and $T_{v'}(G')$ are called equivalent of second degree if and only if there exists a bijective function between their neighbor vertex sets such that if two vertices u and u' from the two neighbor vertex sets are mapped then (i) $T_u(G)$ and $T_{u'}(G')$ are equivalent of first degree, and (ii) there exists another bijective function between neighbor edges of the form (u, u_x) and the neighbor edges of form (u', u'_x) such that $C(u_x) = C(u'_x)$ considering $T_v(G)$ and $T_{v'}(G')$ WLTs.*

Given the equivalent WLTs of first degree of the directed graphs in Figure 10 and Figure 11, we can see the following equivalent WLTs of second degree.

- WLTs $T_{u_1}(G)$, and $T_{v_1}(G')$ are equivalent of second degree.
- WLTs $T_{u_5}(G)$, and $T_{v_5}(G')$ are equivalent of second degree.
- WLTs $T_{u_9}(G)$, and $T_{v_9}(G')$ are equivalent of second degree.
- WLTs $T_{u_{10}}(G)$, and $T_{v_{10}}(G')$ are equivalent of second degree.
- WLTs $T_{u_2}(G)$, and $T_{v_2}(G')$ are equivalent of second degree.
- WLTs $T_{u_6}(G)$, and $T_{v_6}(G')$ are equivalent of second degree.
- WLTs $T_{u_3}(G), T_{u_7}(G), T_{u_{11}}(G), T_{v_3}(G'), T_{v_7}(G')$, and $T_{v_{11}}(G')$ are equivalent of second degree.
- WLTs $T_{u_4}(G), T_{u_8}(G), T_{u_{12}}(G), T_{v_4}(G'), T_{v_8}(G')$, and $T_{v_{12}}(G')$ are equivalent of second degree.

Definition 19 (Equivalence of third degree between two Walk Length Trees of directed graphs)**.** *Two equivalent of second degree Walk Length Trees $T_v(G)$ and $T_{v'}(G')$ are called*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

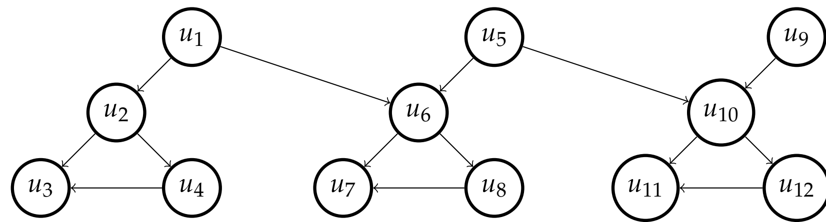

Figure 10. A directed graph G with $n = 12$ and $m = 14$.

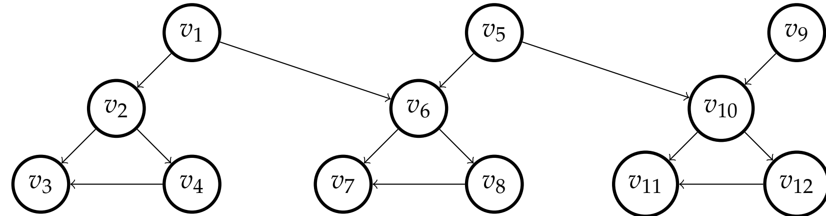

Figure 11. A directed graph G' with $n = 12$ and $m = 14$.

equivalent of third degree if and only if there exists a bijective function between their neighbor vertex sets such that if two vertices u and u' from the two neighbor vertex sets are mapped then (i) $T_u(G)$ and $T_{u'}(G')$ are equivalent of second degree, and

(ii) there exists another bijective function between neighbor edges of the form (u, u_x) and the neighbor edges of form (u', u'_x) such that $C(u_x) = C(u'_x)$ considering $T_v(G)$ and $T_{v'}(G')$ WLTs.

Given the equivalent WLTs of second degree of the directed graphs in Figure 10 and Figure 11, we can see the following equivalent WLTs of third degree.

- WLTs $T_{u_1}(G)$, and $T_{v_1}(G')$ are equivalent of third degree.
- WLTs $T_{u_5}(G)$, and $T_{v_5}(G')$ are equivalent of third degree.
- WLTs $T_{u_9}(G)$, and $T_{v_9}(G')$ are equivalent of third degree.
- WLTs $T_{u_{10}}(G)$, and $T_{v_{10}}(G')$ are equivalent of third degree.
- WLTs $T_{u_2}(G)$, and $T_{v_2}(G')$ are equivalent of third degree.
- WLTs $T_{u_6}(G)$, and $T_{v_6}(G')$ are equivalent of third degree.
- WLTs $T_{u_3}(G)$, and $T_{v_3}(G')$ are equivalent of third degree.
- WLTs $T_{u_7}(G)$, and $T_{v_7}(G')$ are equivalent of third degree.
- WLTs $T_{u_{11}}(G)$, and $T_{v_{11}}(G')$ are equivalent of third degree.
- WLTs $T_{u_4}(G)$, and $T_{v_4}(G')$ are equivalent of third degree.
- WLTs $T_{u_8}(G)$, and $T_{v_8}(G')$ are equivalent of third degree.
- WLTs $T_{u_{12}}(G)$, and $T_{v_{12}}(G')$ are equivalent of third degree.

Following Definition 18 and Figure Definition 19, we can generalize the following definition of WLTs equivalency for directed graphs.

Definition 20 (Equivalence of $(i + 1)$-th degree between two Walk Length Trees of directed graphs)**.** *Two equivalent of i-th degree Walk Length Trees $T_v(G)$ and $T_{v'}(G')$ are called equivalent of $(i + 1)$-th degree, where $i > 0$, if and only if there exists a bijective function between their neighbor vertex sets such that if two vertices u and u' from the two neighbor vertex sets are mapped then (i) $T_u(G)$ and $T_{u'}(G')$ are equivalent of i degree, and (ii) there exists another bijective function between neighbor edges of the form (u, u_x) and the neighbor edges of form (u', u'_x) such that $C(u_x) = C(u'_x)$ considering $T_v(G)$ and $T_{v'}(G')$ WLTs.*

Definition 21 (Equivalence between two Walk Length Trees of directed graphs)**.** *Two equivalent WLTs of degree n-th from the same or two different directed graphs are called equivalent, where n be the number of vertices in the graph.*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Lemma 13 (Time complexity of Algorithm 6)**.** *Algorithm 6 runs in polynomial time with respect to m and n.*

Lemma 14 (Time complexity of obtaining all WLTs of $(i+1)$-th degree)**.** *Suppose, we partition all WLTs of two directed graph such that all equivalent WLTs of i-th degree are in the same part, and any two WLTs from two different parts are not equivalent of degree i. We can further partition each part by applying Algorithm 6 for all pairs, such that all WLTs are equivalent of $(i+1)$-th degree in each new part and any two WLTs from two different new parts are not equivalent of degree $(i+1)$-th in polynomial time with respect to m and n.*

Lemma 15 (On the upper bound of WLTs of i-th degree for directed graphs)**.** *Suppose, we partition all WLTs of two directed graph such that all equivalent WLTs of i-th degree are in the same part, and any two WLTs from two different parts are not equivalent of degree i. Suppose, we further partition each part to obtain equivalent of degree $(i+1)$ by applying Algorithm 6 for all pairs, If after this further partitioning, we observe that any two equivalent WLTs that are equivalent of degree i are still equivalent of degree $(i+1)$, then we can say that any two equivalent WLTs that are equivalent of degree i are equivalent of degree n.*

Proof. For any two equivalent WLTs of i-th degree, the neighbor vertex sets and the corresponding V_A and V_B are fixed in Algorithm 6. So for all pair of equivalent vertices of i-th degree, if Algorithm 6 returns $TRUE$, then any of such pair of WLTs must also return true when we check them for equivalency of any j-th degree, where $j > i$. □

Lemma 16 (Time complexity to check Walk Length Tree equivalency of directed graph)**.** *Suppose, we partition all WLTs of two directed graph such that all equivalent WLTs of i-th degree are in the same part, for $n > i > 0$. It requires polynomial time with respect to m and n.*

3.9. WLT and its equivalent vertices

We need the following information about a WLT $T_v(G)$ to describe our algorithms.

- the graph G,
- reference vertex v,
- a partition of equivalent vertices with respect to $T_v(G)$, such that all vertices from a single part are equivalent and the WLTs where these vertices are reference vertices are equivalent too.
- Moreover two vertices from two different parts are not equivalent with respect to $T_v(G)$ or the WLTs, where these vertices are reference vertices are not equivalent.

For example, for the Walk Length Tree given in Figure 3, we need the following information.

- the graph which is given in in Figure 1.
- reference vertex a,
- the partition of equivalent vertices is: $\{\{b\},\{c,d,e\},\{a\},\{f,g,h\}\}$

And, for the Walk Length Tree given in Figure 4, we need the following information.

- the graph which is given in in Figure 2.
- reference vertex B,
- the partition of equivalent vertices is: $\{\{C\},\{D,E,G\},\{B\},\{A,H,F\}\}$

4. Theorems and proposed algorithms

In this section, we describe some properties and theorems for graphs with automorphism. Based on those theorems, we developed polynomial time algorithms for detecting graph automorphism. Those are also served as the basis for detecting isomorphic graphs. Finally, we propose polynomial time algorithms for graph isomorphism problem too.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021
doi:10.20944/preprints202108.0467.v1

Algorithm 6 Checking WLTs equivalency of $(i+1)$-th degree

Input: $T_v(G), T_{v'}(G')$ two equivalent WLTs of i-th degree

1: **if** the number of neighbor edges or neighbor vertices or host vertices of $T_v(G)$ is not equal to those of $T_{v'}(G')$ **then**
2:     **return** $FALSE$
3: **end if**
4: mark all neighbor vertices of $T_{v'}(G')$ as *unchecked*
5: **for** each neighbor vertex u of $T_v(G)$ **do**
6:     let V_A be the list of host vertices of u that are sorted considering their comprehensive ranks with respect to $T_v(G)$
7:     **for** each *unchecked* neighbor vertex u' of $T_{v'}(G')$ such that $T_u(G)$ and $T_{u'}(G')$ are two equivalent WLTs of i-th degree **do**
8:         let V'_A be the list of host vertices of u' that are sorted considering their comprehensive ranks with respect to $T_{v'}(G')$
9:         $SUCCESS = FALSE$
10:         **if** $|V_A| = |V'_A|$ **then**
11:             $SUCCESS = TRUE$
12:             **for** $j = 0, \ldots, |V_A|$ **do**
13:                 let $v^j_A \in V'_A$ and $v^j_{A'} \in V'_A$
14:                 **if** $C(v^j_A) \neq C(v^j_{A'})$ considering $T_v(G)$ and $T_{v'}(G')$ or the comprehensive rank of the corresponding neighbor edges are not equal considering $T_u(G)$ and $T_{u'}(G')$ **then**
15:                     $SUCCESS = FALSE$
16:                     break
17:                 **end if**
18:             **end for**
19:         **end if**
20:         **if** $SUCCESS = TRUE$ **then**
21:             mark u' as *checked*
22:             break
23:         **end if**
24:     **end for**
25: **end for**
26: **if** all neighbor vertices of $T_{v'}(G')$ are *checked* **then**
27:     **return** $TRUE$
28: **end if**
29: **return** $FALSE$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Theorem 4 (Isomorphic undirected graphs and one to one correspondence between vertices)**.** *Let $G(V,E)$ and $G'(V',E')$ be two isomorphic undirected graphs that are connected. That means there exists a bijective function between V and V' that preserves adjacency relationship. If that bijective function maps between $v \in V$ and $v' \in V'$ then $T_v(G)$ and $T_{v'}(G')$ are equivalent.*

Proof. Suppose, the bijective function maps between $u \in V$ and $u' \in V'$. Both u and u' have equal number of l length paths from v and v' respectively, for $l \geq 1$. Suppose node $N(l,j)$ in $T_v(G)$ contains u in its W set. If u has $a_1,\ldots,a_l$ number of $1,\ldots,l$ length paths from v respectively then u' also has $a_1,\ldots,a_l$ number of $1,\ldots,l$ length paths from v' respectively. So, $N(l,j)$ node in $T_{v'}(G')$ must contain u' in its W set.

First, we claim that these two nodes are shallow equivalent.

- Any vertices in V or V' that has same number of $1,\ldots,l$ length paths from v or v' respectively, like u or u', are the elements of their respective W sets. In both graphs, the number of such vertices are same. So, their W set contains same number of vertices. Their p values are also same, which are $a_1,\ldots,a_l$.
- Their Z contains l radius edges that are incident with one or two vertices from their respective W sets or their other endpoint is in the W set of $N(l,j1)$, where $j1 > j$. In both graphs, the number of such edges are same.

So, these two nodes are shallow equivalent. Thus, it makes these two WLTs shallow equivalent. For every vertex in $G(V,E)$ there must be a vertex in $G'(V',E')$ that satisfy all conditions described in Definition 15 and vice versa. Thus, it makes these two WLTs equivalent. □

Theorem 5 (Isomorphic directed graphs and one to one correspondence between vertices)**.** *Let $G(V,E)$ and $G'(V',E')$ be two isomorphic directed graphs that are weakly connected. That means there exists a bijective function between V and V' that preserves adjacency relationship. If that bijective function maps between $v \in V$ and $v' \in V'$ then $T_v(G)$ and $T_{v'}(G')$ are equivalent.*

Proof. Suppose, the bijective function maps between $u \in V$ and $u' \in V'$. Both u and u' have equal number of l length paths from v and v' respectively, for $l \geq 1$. Suppose node $N(l,j)$ in $T_v(G)$ contains u in its W set. If u has $a_1,\ldots,a_l$ number of $1,\ldots,l$ length paths from v respectively then u' also has $a_1,\ldots,a_l$ number of $1,\ldots,l$ length paths from v' respectively. So, $N(l,j)$ node in $T_{v'}(G')$ must contain u' in its W set. First, we claim that these two nodes are shallow equivalent.

- Any vertices in V or V' that has same number of $1,\ldots,l$ length paths from v or v' respectively, like u or u', are the elements of their respective W sets. In both graphs, the number of such vertices are same. So, their W set contains same number of vertices. Their p values are also same, which are $a_1,\ldots,a_l$.
- Their Z contains l radius edges whose terminal vertices are in their respective W sets. In both graphs, the number of such edges are same.

So, these two nodes are shallow equivalent. Thus, it makes these two WLTs shallow equivalent. For every vertex in $G(V,E)$ there must be a vertex in $G'(V',E')$ that satisfy all conditions described in Definition 17 and vice versa. Thus, it makes these two WLTs equivalent of first degree. Both neighbor vertices and edges of $T_v(G)$ have one to one correspondences with those of $T_{v'}(G')$ respectively. So, these two WLTs are equivalent of second degree. We can apply this argument repeatedly, to conclude that $T_v(G)$ and $T_{v'}(G')$ are equivalent of degree n. □

Lemma 17 (Isomorphic graphs and Walk Length Tree equivalency)**.** *Given two isomorphic graphs G and G', for each WLT of G there exists at least one equivalent WLT of G' and vice versa.*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Lemma 18 (Isomorphic graphs and sets of equivalent Walk Length Trees)**.** *Given two isomorphic graphs G and G', suppose we partition all equivalent WLTs such that all WLTs in one part are equivalent with each other and any two WLTs from two different parts are not equivalent. In each part, we have even number of WLTs and exactly half of the WLTs are from one graph.*

Proof. First, we partition all WLTs from G, such that all WLTs in a part are equivalent and any two WLTs from two different parts are not equivalent. Second, we partition all WLTs from G' in the same way. As G and G' are isomorphic, for each WLT $T_v(G)$ of G there exist at least one equivalent WLT $T_{v'}(G')$ from G'. So, in the partition, for any part, if there exist k WLTs, then in the second partition we must have a part with k WLTs such that these $2k$ WLTs are equivalent with each other. □

Now, we will investigate the mapping among the vertices of a graph that has automorphism.

4.1. Graph Automorphism

Lemma 19 (Automorphism and WLT)**.** *Given a graph $G(V, E)$ with automorphism. Suppose, we can map between u and v. Then, $T_u(G)$ and $T_v(G)$ must be equivalent.*

Proof. Let G' be a copy of the graph G. G and G' must be isomorphic. Then there must be a mapping between u and v', where v' be the copied vertex of v. According to Theorem 4 and Theorem 5, $T_u(G)$ and $T_{v'}(G')$ are equivalent. So, $T_u(G)$ and $T_v(G)$ must be equivalent too. □

Algorithm 7 Graph Automorphism of undirected graph G

Input: $G(V, E)$, A_G
Output: $TRUE$ or $FALSE$

```
1: compute A_G^r , where r = 2, ..., n
2: compute all WLTs of G
3: partition all WLTs, s.t all equivalent WLTs are in one part
4: if every part has only one WLT then
5:     return FALSE
6: end if
7: pick any part that has more than one WLTs
8: possibleMapping := number of WLTs in the chosen part
9: pick any two WLTs T_u(G) and T_v(G) from the chosen part
10: mark all vertices as both unmappedL and unmappedR
11: MAP^undir_automorphism(T_u(G), T_v(G))
12: while there is unmappedL vertices do
13:     choose a set of equivalent unmappedL vertices A and unmappedR vertices B with
        respect to T_u(G) and T_v(G) respectively, s.t all WLTs are equivalent where these
        vertices are reference vertices.
14:     for each unmappedL vertex u_a ∈ A do
15:         choose an unmappedR vertex v_a ∈ B
16:         MAP^undir_automorphism(T_{u_a}(G), T_{v_a}(G))
17:     end for
18:     possibleMapping := possibleMapping ∗ |A|
19: end while
20: return TRUE
```

Lemma 20 (No automorphism)**.** *Given a graph $G(V, E)$, we partition all WLTs, so that each part contains equivalent WLTs and two WLTs from two different parts, are not equivalent. If each part contains exactly one WLT, we can conclude that G does not have any automorphism.*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Proof. We prove it by contradiction. Suppose G has automorphism and $u \in V$ maps to $v \in V$. So, $T_u(G)$ and $T_v(G)$ are equivalent and they are kept in the same part, which creates the contradiction. □

Algorithm 8 ROUTINE $MAP^{undir}_{automorphism}(T_u(G), T_v(G))$

1: declare: u maps to v
2: mark u and v as *mappedL* and *mappedR* respectively. They are no more marked as *unmappedL* or *unmappedR*
3: **while** in $T_u(G)$, there is an *unmappedL* vertex u_a, for which there is no equivalent *unmappedL* vertex (w.r.t $T_u(G)$) s.t their WLTs are equivalent **do**
4: let v_b be the equivalent *unmappedR* vertex in $T_v(G)$ s.t $T_{u_a}(G)$ and $T_{v_b}(G))$ are equivalent
5: $MAP_{automorphism}(T_{u_a}(G), T_{v_b}(G))$
6: **end while**

Lemma 21 (Automorphism)**.** *Given a graph $G(V, E)$, we partition all WLTs, so that each part contains equivalent WLTs and two WLTs from two different parts, are not equivalent. Suppose, one part contains more than one WLTs. We take two of such equivalent WLTs: $T_u(G)$ and $T_v(G)$, we claim that G has automorphism and we can map between u and v.*

Proof. We can reconstruct the same graph from $T_u(G)$ and $T_v(G)$ separately. Suppose, we create leveled graphs from these two WLTs. Consider, the leveled graphs are isomorphic for all levels $1, \ldots, l-1$ and they are not isomorphic at level l, where $l < n$. In level l, we have at least one edge in one leveled graph which is different from other. We claim that it is not possible. According to the definition of leveled graph, it gets all edges of j-radius from the reference vertex at level $j = 1, \ldots, n$. So, if the leveled graphs are not isomorphic at level l, it is not possible to have them isomorphic at level $l+1, \ldots, n$. This is not possible, as it is the same graph. The level graphs at n-th level must be isomorphic. So, both leveled graphs from these two WLTs are isomorphic at any level. Thus, the graph has automorphism. It also proves that there exists a valid map between u and v. □

Lemma 22 (Mapping in automorphism)**.** *Given a graph $G(V, E)$ with automorphism, we partition all WLTs, so that each part contains equivalent WLTs and two WLTs from two different parts, are not equivalent. Suppose, one part contains more than one WLTs. We take two of such equivalent WLTs: $T_u(G)$ and $T_v(G)$. Let there exist two vertices $u_a, u_b \in V$ such that they are equivalent with respect to $T_u(G)$ and $T_v(G)$. Let u_a and u_b be from $T_u(G)$ and $T_v(G)$ respectively. Let $T_{u_a}(G)$ and $T_{u_b}(G)$ be equivalent and there exists no such vertex pair. If we map between u and v, we must map between u_a and u_b.*

Proof. Like the proof of Lemma 21, we reconstruct G from $T_u(G)$ and $T_v(G)$ separately. Both leveled graphs from these two WLTs are isomorphic at any level. As u_a and u_b are equivalent with respect to $T_u(G)$ and $T_v(G)$, the edges, that are incident to them, are adding with the leveled graph exactly the same way. So, u_a must map to u_b if u maps to v. □

Lemma 23 (A vertex with no equivalent vertex)**.** *Given a graph $G(V, E)$ with automorphism, we partition all WLTs, so that each part contains equivalent WLTs and two WLTs from two different parts, are not equivalent. Suppose, one part contains more than one WLTs. We take two of such equivalent WLTs: $T_u(G)$ and $T_v(G)$. In both WLTs, there exists a vertex $u_a \in V$ such that it is equivalent with respect to $T_u(G)$ and $T_v(G)$. And there is no other vertices that is*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Algorithm 9 Graph Automorphism of directed graph G

Input: $G(V, E)$, A_G
Output: $TRUE$ or $FALSE$

1: compute A^r_G , where $r = 2, \ldots, n$
2: compute all WLTs of G
3: partition all WLTs, s.t all equivalent WLTs are in one part
4: **if** every part has only one WLT **then**
5:    **return** $FALSE$
6: **end if**
7: $possibleMapping := 1$
8: mark all vertices as both *unmappedL* and *unmappedR*
9: mark all WLTs as *incomplete*
10: create a stack, where each item is an ordered pair of vertices
11: choose any part that has more than one WLTs. choose any two WLTs $T_{u_a}(G)$ and $T_{u_b}(G)$ from the part. push (u_a, u_b) in the stack
12: $possibleMapping := possibleMapping*$(the number of WLTs in the chosen part-1)
13: **while** the stack is not empty **do**
14:    $SUCCESS = FALSE$
15:    **while** $SUCCESS == FALSE$ and the stack is not empty **do**
16:       pop from stack. let (u, v) be the pair from the stack.
17:       **if** both $T_u(G)$ and $T_v(G)$ are marked as *incomplete* **then**
18:          $SUCCESS = TRUE$
19:       **end if**
20:    **end while**
21:    **if** $SUCCESS == FALSE$ **then**
22:       $BREAK$
23:    **end if**
24:    $MAP^{dir}_{automorphism}(T_u(G), T_v(G))$
25:    **while** there is *unmappedL* vertices in $T_u(G)$ **do**
26:       choose a set of equivalent *unmappedL* vertices A and *unmappedR* vertices B with respect to $T_u(G)$ and $T_v(G)$ respectively, s.t all WLTs are equivalent where these vertices are reference vertices.
27:       **for** each *unmappedL* vertex $u_a \in A$ **do**
28:          choose an *unmappedR* vertex $v_a \in B$
29:          $MAP^{dir}_{automorphism}(T_{u_a}(G), T_{v_a}(G))$
30:       **end for**
31:       $possibleMapping := possibleMapping * |A|$
32:    **end while**
33:    mark $T_u(G)$ and $T_v(G)$ as *complete*
34: **end while**
35: **for** each *unmappedL* vertex u **do**
36:    declare: u maps to u
37: **end for**
38: **return** $TRUE$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Algorithm 10 ROUTINE $MAP^{dir}_{automorphism}(T_u(G), T_v(G))$

1: **if** u is already mapped to v **then**
2: $RETURN$
3: **end if**
4: declare: u maps to v
5: mark u and v as $mappedL$ and $mappedR$ respectively. they are no more marked as $unmappedL$ or $unmappedR$
6: partition all *incomplete* WLTs, whose reference vertices are not in the stack and are neighbor vertices of u and v, s.t each part contain equivalent WLTs and two WLTs from two different parts are not equivalent.
7: **for** each part that contains more than one WLTs **do**
8: $possibleMapping := possibleMapping * 0.5*$the number of WLTs in the part
9: create ordered pairs of vertices with the reference vertices of the WLTs in the part s.t in each ordered pair (u_c, u_d), u_c and u_d are neighbor vertices of u and v respectively. and each vertex participates in one of such pair only. push all such pairs in the stack.
10: **end for**
11: **while** in $T_u(G)$, there is an $unmappedL$ vertex u_a, for which there is no equivalent $unmappedL$ vertex (w.r.t $T_u(G)$) s.t their WLTs are equivalent **do**
12: let v_b be the equivalent $unmappedR$ vertex in $T_v(G)$ s.t $T_{u_a}(G)$ and $T_{v_b}(G))$ are equivalent
13: $MAP^{dir}_{automorphism}(T_{u_a}(G), T_{v_b}(G))$
14: **end while**

equivalent to u_a with respect to $T_u(G)$ and $T_v(G)$. If we map between u and v, u_a must map to itself.

Proof. It follows from Lemma 22. □

Lemma 24 (Further mappings in Automorphism)**.** *Given a graph $G(V, E)$ with automorphism, we partition all WLTs, so that each part contains equivalent WLTs and two WLTs from two different parts, are not equivalent. Suppose, one part contains more than one WLTs. We take two of such equivalent WLTs: $T_u(G)$ and $T_v(G)$. Let there exist two vertices $u_a, u_b \in V$ such that (i) they are equivalent with respect to $T_u(G)$ and $T_v(G)$ and (ii) $T_{u_a}(G)$ and $T_{u_b}(G)$ are equivalent. Let u_a and u_b be from $T_u(G)$ and $T_v(G)$ respectively. If we map between u and v, there exists a valid mapping between u_a and u_b.*

Proof. Like the proof of Lemma 21, we reconstruct G from $T_u(G)$ and $T_v(G)$ separately. Both leveled graphs from these two WLTs are isomorphic at any level. The edges from any equivalent vertex pair with respect to $T_u(G)$ and $T_v(G)$ like u_a and u_b, that are incident to them, are adding with the leveled graph exactly the same way. As $T_{u_a}(G)$ and $T_{u_b}(G)$ are equivalent, u_a has a valid mapping with v_b. So, if we map u with v, we can map u_a with u_b. □

Remark 2 (Every vertex must be mapped in a graph with automorphism)**.** *Given a graph $G(V, E)$ with automorphism, we partition all WLTs, so that each part contains equivalent WLTs and two WLTs from two different parts, are not equivalent. Suppose, one part contains more than one WLTs. We take two of such equivalent WLTs: $T_u(G)$ and $T_v(G)$. Every vertex must be mapped with another vertex or itself. Suppose, we map u with v. We have the following mappings: (i) for each vertex in $T_u(G)$, there must exist an equivalent vertex with respect to $T_u(G)$ and $T_v(G)$ such that the WLTs, where they are the reference vertices, must be equivalent, (ii) if any two vertices are mapped they must be equivalent with respect to $T_u(G)$ and $T_v(G)$ and the WLTs where they are the reference vertices must be equivalent, (iii) if two vertices are*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Algorithm 11 Graph Isomorphism (GI) between undirected graph G, and G'

Input: two graphs and their adjacency matrix: $G(V,E)$, $G'(V',E')$, A_G, and $A_{G'}$
Output: $TRUE$ or $FALSE$

```
1: compute A^r_G and A^r_G', where r = 2,...,n
2: compute all WLTs of both G and G'
3: partition all WLTs s.t all equivalent WLTs are in one part
4: for each part do
5:    if the number of WLTs in the part is odd then
6:       return FALSE
7:    else if more than half of the WLTs are from one of the graph then
8:       return FALSE
9:    end if
10: end for
11: for each part that has only two WLTs: T_{v_z}(G) and T_{v_z'}(G'), s.t v_z is not mapped to
    v_z' do
12:    if MAP_undir(T_{v_z}(G), T_{v_z'}(G')) returns FALSE then
13:       return FALSE
14:    end if
15: end for
16: pick any two equivalent WLTs, s.t their reference vertices are mapped to each other
17: if no such pair of WLTs are found in the previous statement then
18:    pick any pair of equivalent WLTs
19:    if no such pair found then
20:       return FALSE
21:    end if
22: end if
23: Let T_{v_i}(G) and T_{v_i'}(G') be the picked WLTs.
24: if v_i is not already mapped to v_i' then
25:    if MAP_undir(T_{v_i}(G), T_{v_i'}(G')) returns FALSE then
26:       return FALSE
27:    end if
28: end if
29: while there is unmapped vertices in G do
30:    choose a set of equivalent unmapped vertices A and unmapped vertices B with
       respect to T_{v_i}(G) and T_{v_i'}(G') respectively, s.t all WLTs are equivalent where these
       vertices are reference vertices.
31:    if |A| ≠ |B| then
32:       return FALSE
33:    end if
34:    for each unmapped vertex u_a ∈ A do
35:       choose an unmapped vertex u'_a ∈ B
36:       if no such vertex found in B then
37:          return FALSE
38:       end if
39:       MAP_undir(T_{u_a}(G), T_{u'_a}(G'))
40:    end for
41: end while
42: permute both A_G and A_G' based on vertex mappings declared in MAP_undir() routine
43: if A_G and A_G' are same then
44:    return TRUE
45: end if
46: return FALSE
```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Algorithm 12 ROUTINE $MAP_{undir}(T_{v_j}(G), T_{v_{j'}}(G'))$

1: **if** $T_{v_j}(G)$ and $T_{v_{j'}}(G')$ are not equivalent **then**
2: **return** $FALSE$
3: **end if**
4: declare: v_j maps to $v_{j'}$
5: **while** in $T_{v_j}(G)$, there is an unmapped vertex v_z, for which there is no equivalent unmapped vertex (w.r.t $T_{v_j}(G)$) s.t their WLTs are equivalent **do**
6: let $v_{z'}$ be the equivalent unmapped vertex w.r.t $T_{v_{j'}}(G')$, s.t $T_{v_z}(G)$ and $T_{v_{z'}}(G')$ are equivalent
7: **if** there exist more than one such vertex in $T_{v_{j'}}(G')$, **then**
8: **return** $FALSE$
9: **end if**
10: **if** $MAP(T_{v_z}(G), T_{v_{z'}}(G'))$ returns $FALSE$ **then**
11: **return** $FALSE$
12: **end if**
13: **end while**
14: **return** $TRUE$

equivalent with respect to $T_u(G)$ *and* $T_v(G)$ *and the WLTs where they are the reference vertices are equivalent then those vertices can be mapped, (iv) if we partition all vertices in* $T_u(G)$ *and* $T_v(G)$ *such that all vertices in a part are equivalent with respect to* $T_u(G)$ *and* $T_v(G)$ *and the WLTs, where they are reference vertices, are equivalent. In each part, there are even number of vertices and exactly half of the vertices are from* $T_u(G)$.

For example, all eight WLTs in graph of Figure 1 are equivalent. Below, we are writing each WLT with its reference vertices and their vertex partition as described in subsection 3.9.

- WLT with a as the reference vertex. The vertex partition is: $\{\{b\}, \{c, d, e\}, \{a\}, \{f, g, h\}\}$.
- WLT with b as the reference vertex. The vertex partition is: $\{\{a\}, \{f, g, h\}, \{b\}, \{c, d, e\}\}$.
- WLT with c as the reference vertex. The vertex partition is: $\{\{g\}, \{a, d, e\}, \{c\}, \{b, f, h\}\}$.
- WLT with d as the reference vertex. The vertex partition is: $\{\{f\}, \{a, c, e\}, \{d\}, \{b, g, h\}\}$.
- WLT with e as the reference vertex. The vertex partition is: $\{\{h\}, \{a, c, d\}, \{e\}, \{b, f, g\}\}$.
- WLT with f as the reference vertex. The vertex partition is: $\{\{d\}, \{b, g, h\}, \{f\}, \{a, c, e\}\}$.
- WLT with g as the reference vertex. The vertex partition is: $\{\{c\}, \{b, f, h\}, \{g\}, \{a, d, e\}\}$.
- WLT with h as the reference vertex. The vertex partition is: $\{\{e\}, \{b, g, f\}, \{h\}, \{a, c, d\}\}$.

Let we map a with g. Now consider the vertex partitions from both WLTs: $\{\{b\}, \{c, d, e\}, \{a\}, \{f, g, h\}\}$ and $\{\{c\}, \{b, f, h\}, \{g\}, \{a, d, e\}\}$. The following pair of parts contain equivalent vertices considering their respective WLTs:

- $\{b\}$ and $\{c\}$.
- $\{c, d, e\}$ and $\{b, f, h\}$.
- $\{a\}$ and $\{g\}$
- $\{f, g, h\}$ and $\{a, d, e\}$.

So, b must map to c. Any vertex in part $\{c, d, e\}$ can be mapped to any vertex in $\{b, f, h\}$. We will assign one by one and apply the mappings that becomes obvious. Let c maps

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

Algorithm 13 Graph Isomorphism (GI) between directed graph G, and G'

Input: two graphs and their adjacency matrix: $G(V,E)$, $G'(V',E')$, A_G, and $A_{G'}$
Output: $TRUE$ or $FALSE$

1: compute A^r_G and $A^r_{G'}$, where $r = 2,\ldots,n$. compute all WLTs of both G and G' and all WLTs as *incomplete*. partition all WLTs, s.t all equivalent WLTs are in one partition and no two WLTs from two different parts are not equivalent
2: **if** any part has odd number of WLTs or more than half of its WLTs are from one of the graph **then**
3:    **return** $FALSE$
4: **end if**
5: mark all vertices as *unmapped* and mark all WLTs as *incomplete*. create a stack, where each item is an ordered pair of vertices. choose any part that has more than one WLTs. choose any two WLTs $T_u(G)$ and $T_{u'}(G')$ from the part. push (u,u') in the stack
6: **if** $MAP_{dir}(T_u(G),T_{u'}(G'))$ returns $FALSE$ **then**
7:    **return** $FALSE$
8: **end if**
9: **while** the stack is not empty **do**
10:    $SUCCESS = FALSE$
11:    **while** $SUCCESS == FALSE$ and the stack is not empty **do**
12:       pop from stack. let (v,v') be the pair from the stack.
13:       **if** both $T_v(G)$ and $T_{v'}(G')$ are marked as *incomplete* **then**
14:          $SUCCESS = TRUE$
15:       **end if**
16:    **end while**
17:    **if** $SUCCESS == FALSE$ **then**
18:       $BREAK$
19:    **end if**
20:    **while** there is *unmapped* vertices in $T_v(G)$ **do**
21:       create two equivalent and *unmapped* vertex sets A and B w.r.t $T_v(G)$ and $T_{v'}(G')$ respectively, s.t all WLTs are equivalent where these vertices are reference vertices.
22:       **if** $|A| \neq |B|$ **then**
23:          **return** $FALSE$
24:       **end if**
25:       **for** each *unmapped* vertex $u_a \in A$ **do**
26:          choose an *unmapped* vertex $u_{a'} \in B$
27:          **if** no such vertex found in B **then**
28:             **return** $FALSE$
29:          **end if**
30:          **if** $MAP_{dir}(T_{u_a}(G),T_{u_{a'}}(G'))$ returns $FALSE$ **then**
31:             **return** $FALSE$
32:          **end if**
33:       **end for**
34:    **end while**
35:    mark $T_{u_a}(G)$ and $T_v(G)$ as *complete*
36: **end while**
37: permute both A_G and $A_{G'}$ based on vertex mappings declared in $MAP_{dir}()$ routine
38: **if** A_G and $A_{G'}$ are same **then**
39:    **return** $TRUE$
40: **end if**
41: **return** $FALSE$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

to h. Consequently, g maps to e. Then, we can map d to b. Thus, f maps to a. Finally, e and h map to f and d respectively. So, one mapping among the vertices in this graph can be shown by the following two lists: $[a, b, c, g, d, f, e, h]$ maps to $[g, c, h, e, b, a, f, d]$. The i-th vertex in the first list maps to the i-th vertex in the second list, where $0 \leq i < 8$. Consider, any pair of vertices, where we have a mapping from these lists. Let, b and c. We can verify this mapping from the two WLTs, where they are the reference vertices. In the first WLT (where reference vertex is b) the i-th vertex from the first list is equivalent with the i-th vertex of the second list in the second WLT (where reference vertex is c) considering their respective WLTs. This is true for any pair of vertices that has mapping. Now, we need to count how many ways the vertices of a graph that has automorphism can be mapped.

Algorithm 14 ROUTINE $MAP_{dir}(T_{v_j}(G), T_{v_{j'}}(G'))$

1: **if** $T_{v_j}(G)$ and $T_{v_{j'}}(G')$ are not equivalent **then**
2:    **return** $FALSE$
3: **end if**
4: declare: v_j maps to $v_{j'}$
5: mark both v_j and $v_{j'}$ as *mapped*
6: partition all *incomplete* WLTs, whose reference vertices are neighbor vertices of v_j and $v_{j'}$ and their reference vertices are not in the stack, s.t each part contains equivalent WLTs and two WLTs from two different parts are not equivalent.
7: **for** each part **do**
8:    **if** the part contains odd number of WLTs or more than half of WLTs are from one graph **then**
9:       **return** $FALSE$
10:    **end if**
11:    create ordered pairs of vertices with the reference vertices of the WLTs in the part s.t in each ordered pair (u, u'), u and u' are neighbor vertices of v_j and $v_{j'}$ respectively and each vertex participates in one of such pair only. push all such pairs in the stack.
12: **end for**
13: **while** in $T_{v_j}(G)$, there is an *unmapped* vertex u_a, for which there is no equivalent *unmapped* vertex (w.r.t $T_{v_j}(G)$) s.t their WLTs are equivalent **do**
14:    let $u_{a'}$ be the equivalent *unmapped* vertex in $T_{v_{j'}}(G')$ s.t $T_{u_a}(G)$ and $T_{u_{a'}}(G')$ are equivalent
15:    **if** no such vertex found **then**
16:       **return** $FALSE$
17:    **end if**
18:    **if** $MAP_{dir}(T_{u_a}(G), T_{u_{a'}}(G'))$ returns $FALSE$ **then**
19:       **return** $FALSE$
20:    **end if**
21: **end while**
22: **return** $TRUE$

Theorem 6 (Time complexity for detecting graph automorphism)**.** *Graph Automorphism is in P.*

Proof. Both Algorithm 7 and Algorithm 9 run in polynomial time with respect to n and m. Both algorithms are written according to the lemmas that we described in this subsection. Both algorithms can detect graphs with automorphism. It also can show a valid mapping among the vertex set. It also find the number of valid mappings (denoted as $possibleMapping$ in the algorithms) automorphism. It is easy to show all possible mapping among the vertices by modifying the algorithm. In case of directed graph, we need to keep track of the neighbor vertices once the host vertices are mapped. We use a

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

stack in Algorithm 9 for this purpose. Thus, we prove that graph automorphism is in P. □

4.2. Graph Isomorphism

Let $G(V, E)$ and $T_{u'}(G')$ be two isomorphic graphs. We can make the following remarks on the WLTs of $G(V, E)$ and $G'(V', E')$.

Remark 3. *All WLTs of both graph can be partitioned in such a way that in all equivalent WLTs are in the same part. Any two WLTs from two different parts are not equivalent. Each part contains even number of WLTs and exactly half of WLTs are from one graph. We take two equivalent WLTs: $T_u(G)$ and $T_{u'}(G')$ from such a part. Every vertex in $T_u(G)$ must be mapped to a vertex in $T_{u'}(G')$ and u can be mapped to u'. Suppose, we map u with u'. We have the following mappings: (i) for each vertex in $T_u(G)$, there must exist an equivalent vertex with respect to $T_u(G)$ and $T_{u'}(G')$ such that the WLTs, where they are the reference vertices, must be equivalent, (ii) if any two vertices are mapped they must be equivalent with respect to $T_u(G)$ and $T_{u'}(G')$ and the WLTs where they are the reference vertices must be equivalent, (iii) if two vertices are equivalent with respect to $T_u(G)$ and $T_{u'}(G')$ and the WLTs where they are the reference vertices are equivalent then those vertices can be mapped, (iv) if we partition all vertices in $T_u(G)$ and $T_{u'}(G')$ such that all vertices in a part are equivalent with respect to $T_u(G)$ and $T_{u'}(G')$ and the WLTs, where they are reference vertices, are equivalent. In each part, there are even number of vertices and exactly half of the vertices are from $T_u(G)$, (v) If we*

We can validate all statements in Remark 3 in the following way. $G'(V', E')$ can be a graph that is created by changing the vertex labels of $G(V, E)$. So, Remark 3 is extended from Remark 2.

Lemma 25 (Correctness of algorithms for graph isomorphism)**.** *Algorithm 11 and Algorithm 13 can detect isomorphic and non-isomorphic graphs correctly.*

Proof. Algorithm 11 and Algorithm 13 are written based on the similar principle like in Algorithm 7 and Algorithm 9 respectively. So these algorithms can detect isomorphic graphs correctly. So, the permutation of adjacency matrices yields the same matrix. On the other hand, if non-isomorphic graphs are given as input to these algorithms. It can detect it correctly too. Because, no permutation of their adjacency matrices can make them equal. □

Lemma 26 (Time complexity for detecting graph isomorphism)**.** *Algorithm 13 and Algorithm 11 runs in polynomial time with respect to m and n.*

Theorem 7 (Time complexity for detecting graph isomorphism)**.** *Graph isomorphism is in P.*

Proof. It follows from Lemma 25 and Lemma 26. □

5. Conclusion

Thus, we prove that both graph isomorphism and automorphism problems can be solved in polynomial time.

Funding: The author received no funding for this work.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: The author would like to thank Dr. Mohammad Parvez (M.Parvez@qu.edu.sa) for reviewing the draft of this paper several times with patience.

Conflicts of Interest: The author declares no conflict of interest.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 August 2021 doi:10.20944/preprints202108.0467.v1

References

1. Luks, Eugene M. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of computer and system sciences* **1982**, 25-1, Elsevier, 42–65.
2. Babai, László; Kantor, William M.; Luks, Eugene M. Computational complexity and the classification of finite simple groups. 24th Annual Symposium on Foundations of Computer Science, IEEE, **1983** 162–171.
3. Hopcroft, JE. Linear time algorithm for isomorphism of planar graphs. ACM symposium on Theory of computing (STOC), **1974** 172–184.
4. Lueker, George S.; Booth Kellogg S. A linear time algorithm for deciding interval graph isomorphism. *Journal of the ACM (JACM)* **1979**, 26-2, ACM, 183–195.
5. Colbourn, Charles J. On testing isomorphism of permutation graphs. *Networks* **1981**, 11-1, Wiley Online Library, 13–21.
6. McKay, Brendan D. Computing automorphisms and canonical labellings of graphs. *Combinatorial mathematics*, Springer, **1978**, 223–232.
7. Muzychuk, M. A solution of the isomorphism problem for circulant graphs. Proceedings of the London Mathematical Society, 88-1, Wiley Online Library, **2004**, 1–41.
8. Bodlaender, Hans L. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *Journal of Algorithms*, 11-4, **1990**, Elsevier, 631–643.
9. Dasgupta, Sanjoy; Papadimitriou Christos H.; Vazirani, Umesh Virkumar. *Algorithms*, McGraw-Hill Higher Education New York.
10. Miller, Gary. Isomorphism testing for graphs of bounded genus. Proceedings of the twelfth annual ACM symposium on Theory of computing, **1980**, 225–235.
11. Babai, László, Grigoryev, D Yu; Mount, David M. Isomorphism of graphs with bounded eigenvalue multiplicity, Proceedings of the fourteenth annual ACM symposium on Theory of computing, **1982**, 310–324.
12. Babai, László. Graph Isomorphism in Quasipolynomial Time. arXiv, 1512.03547, cs.DS.
13. McKay, Brendan D.; Piperno, Adolfo. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60, Elsevier, 2014, 94–112.
14. Gallian, Joseph A. Graph labeling. *The electronic journal of combinatorics*, **2012**, DS6–Dec.
15. McKay, Brendan D. Backtrack programming and isomorph rejection on ordered subsets, *Ars Combinatoria*, 5, Department of Combinatorics and Optimization, University of Waterloo, **1978**, 65–99.
16. Darga, Paul T.; Liffiton, Mark H.; Sakallah, Karem A.; Markov, Igor L. Exploiting structure in symmetry detection for CNF. Proceedings of the 41st annual Design Automation Conference, **2004**, 530–534.
17. Darga, Paul T.; Sakallah, Karem A.; Markov, Igor L. Faster symmetry discovery using sparsity of symmetries, 45th ACM/IEEE Design Automation Conference, **2008**, 149–154.
18. Junttila, Tommi; Kaski, Petteri. Conflict propagation and component recursion for canonical labeling, International Conference on Theory and Practice of Algorithms in (Computer) Systems, **Springer**, 2011, 151–162.
19. Foggia, Pasquale; Sansone, Carlo; Vento, Mario. A performance comparison of five algorithms for graph isomorphism, Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition, **2001**, 188–199.
20. Neuen, Daniel. Graph isomorphism for unit square graphs, arXiv preprint arXiv:1602.08371, **2016**.
21. Rosen, Kenneth H. *Discrete mathematics and its applications*, 7th Edition, McGraw-Hill, **2012**.
22. Aho, Alfred V.; Hopcroft, John E. *The design and analysis of computer algorithms*, Pearson Education India, **1974**.
23. The Nauty Traces page. Available online: http://cs.anu.edu.au/ bdm/nauty/ (accessed on 18 August 2021).