AN APPROACH TO GRAPH ISOMORPHISM USING SPANNING TREES GENERATED BY BREADTH FIRST SEARCH

Submitted in partial fulfillment of the requirements for the degree of Master of Science

by

ALEXEY ILCHENKO

Department of Mathematics
CASE WESTERN RESERVE UNIVERSITY
August, 2014
We hereby approve the thesis/dissertation of
Alexey Ilchenko

candidate for the degree of
Master of Science*

Committee Chair
Erkki Somersalo

Committee Member
Erkki Somersalo

Committee Member
Daniela Calvetti

Committee Member
John Duncan

Date of Defense
May 16, 2014

*We also certify that written approval has been obtained for any proprietary material contained therein.
Contents

List of Figures 2

List of Tables 3

1 Introduction 5

2 Graph traversals and trees 6

2.1 Trees ................................................. 6
2.2 Breadth First Search .................................. 6
2.3 Spanning Trees ...................................... 9

3 New Algorithm 10

3.1 Shortest distance .................................... 10
3.2 Guessing ............................................. 12
3.3 Backtracking ......................................... 13
3.4 Illustration of the algorithm as a path through a tree ............... 15
3.5 The Algorithm ....................................... 16
3.6 Program checking .................................... 20

4 Random Graphs 21

4.1 Introduction to Erdös-Renyi model ....................... 21
4.2 Generation of Permutation Matrices ..................... 22
4.3 Growth of Random Graphs .............................. 23

5 Experimental Running Times 24

5.1 Introduction to results ................................ 24
5.2 Running times when \( \lambda = 0.5 \) ....................... 24
5.3 Running times when \( \lambda = 2 \) ....................... 25
5.4 Running times when \( \lambda = 16 \) ...................... 26
5.5 Running times when \( \lambda = 128 \) ................. 26
5.6 Running times when \( \lambda \) unbounded and \( p = 1/2 \) .............. 27

6 Discussion 30

7 References 32
List of Figures

1. Sample tree ........................................... 6
2. Sample graph $S$ ....................................... 8
3. Sample spanning trees of graph $S$ .................... 9
4. Spanning tree generated by Breadth First Search .... 10
5. Spanning trees of the same height and width at each level 13
6. All paths in the decision tree ........................ 15
7. One particular path .................................. 16
8. Path with backtracking ............................... 16
9. Graph 1: Log-log plot of data in Table 9 (with $\lambda$ unbounded and $p = 1/2$) 29
10. Graph 2: Log-log plot of data in Table 10 (with $\lambda$ unbounded and $p = 1/2$) 30
## List of Tables

1. Test on Isomorphic graphs with $\lambda = 0.5$ .................................................. 24
2. Test on Non-isomorphic graphs with $\lambda = 0.5$ .................................................. 25
3. Test on Isomorphic graphs with $\lambda = 2$ ............................................................. 26
4. Test on Non-isomorphic graphs with $\lambda = 2$ ..................................................... 26
5. Test on Isomorphic graphs with $\lambda = 16$ ............................................................ 27
6. Test on Non-isomorphic graphs with $\lambda = 16$ ...................................................... 27
7. Test on Isomorphic graphs with $\lambda = 128$ .......................................................... 27
8. Test on Non-isomorphic graphs with $\lambda = 128$ ................................................... 28
9. Test on Isomorphic graphs with $\lambda$ unbounded and $p = 1/2$ ................................. 28
10. Test on Non-isomorphic graphs with $\lambda$ unbounded and $p = 1/2$ ......................... 29
An approach to Graph Isomorphism using

Spanning Trees generated by Breadth First Search

Author ALEXEY ILCHENKO

Abstract

Graph Isomorphism is a problem of determining whether or not a bijective function between two graphs exists which also preserves the adjacency relation between nodes. If graphs, regardless of how they are drawn or in what order the nodes are listed, are shown to be isomorphic, then they are essentially the same. This has direct applications in fields even outside of computer science and mathematics. This paper presents a novel way to determine whether this relationship between any given unweighted and undirected graphs exists by employing the invariant property of shortest distance which we exploit by generating a spanning tree using Breadth First Search. The Algorithm relies on guessing and backtracking, but the number of guesses which are available to be made are heavily reduced from a naive approach. The running times of the presented algorithm are measured by exploring some growth parameters of random graphs.
1 Introduction

A graph $G$ is a collection of nodes $V = \{v_1, v_2, \ldots\}$ together with edges between pairs of nodes $E = \{e_1, e_2, \ldots\}$. In this paper we consider graphs made of a finite number of nodes with edges unweighted and undirected and we start by considering connected graphs, but the corresponding code is also generalized to disconnected graphs. In other words, we talk of graphs with uniquely labeled $n$ nodes for some finite number $n$; we may write $|G| = n$ as the notation here will focus primarily on nodes, not edges so the edge between any two nodes either exists or not. Between some pairs of nodes we find unweighted and undirected edges. Hence, every node $u \in G$ has some set of adjacent nodes $\{v_1, v_2, \ldots, v_k\}$ which are connected by an an edge with $u$. There is always a path from any node $u$ to any other node $v$ in a connected graph. Two graphs $G$ and $G'$ are said to be isomorphic if there exists a bijective function $f$ between the nodes such that if $u$ and $v$ in $G$ are adjacent, then $f(u)$ and $f(v)$ in $G'$ are adjacent. In this way, the mapping $f$ between $G$ and $G'$ preserves edges.

Graphs are encountered in many aspects of life even beyond computer science and mathematics. Anything which exists as a network of points and connections between points can be modeled by a graph. As an emerging example, we can model social interaction and social networks online as graphs. People could be the nodes while the friend relationship between the people could be the edges between nodes. Molecules can be modeled as graphs. Each atom can be viewed as a node and the bonds between as edges. In fact, this is exactly the field of chemical graph theory [3]. Circuit elements form a graph. Resistors, capacitors, LED’s and the like are nodes where electrical wires connecting them are edges. All of these examples can be abstracted as a graph. Only work required is to decide how to work with non-unique points (for example, all resistors of the same resistance are alike) or how to differentiate between unlike edges (for example, we might have to specify that a certain edge represents a double bond or a triple bond in chemical graph theory).

Once we start working with unweighted and undirected graphs, we might ask ourselves whether two graphs that we somehow encounter are essentially the same. Chemical researchers, for example, might store information about large proteins as graphs constructed by an adjacency matrix. However, since there is no canonical atom to start the matrix with, researchers might come up with two different adjacency matrices (or two different graphs) which describe the same large protein (this lack of canonical matrix per graph is described fully within the paper). These two graphs would have to be tested for isomorphism (or non-isomorphism) to determine if they describe the same object, the same large protein but only "viewed" from a different angle.
2 Graph traversals and trees

2.1 Trees

Trees are graphs which contain no cycles, i.e., starting from any node $u$ in a tree, there is no path back to $u$ that does not visit an already visited node (other than the trivial path where the starting and ending node is the same). Consider the following tree.

The node $a$ is called the root of the tree (we can say that the above tree is rooted at $a$). Nodes $c$ and $b$ are called children of the node $a$. Nodes $a$ and $c$ are adjacent because they are connected by an edge. The distance between $f$ and $a$ is 2 since the path from $f$ to $a$ contains 2 edges. The longest path to a node from the root node defines the height of the tree. For the above tree, the longest path from the root node is to node $e$ and is of distance 3. So the height of the tree is defined to be 3. For convenience, in this paper we define the width of the tree at distance $k$ (or level $k$) to be the number of nodes distance $k$ from the root node. For example, the width of the tree at distance 2 is 3 because the only nodes $g$, $d$, and $f$ are a distance 2 from the root $a$. For more complete introduction to trees, graphs, and the Breadth First Search algorithm (central to the next section) see [4].

2.2 Breadth First Search

Breadth First Search (BFS) is an algorithm for traversing graphs. Starting at a node $u$ in $G$, its uses include finding all other nodes (within connected graphs) and also finding a shortest path from $u$ to all other nodes in $G$ (keep in mind that although a shortest path may not be unique, the shortest distance is well-defined). In order to use BFS, we have to pick some node in $G$ and call it the main node. Let us call the main node $u_1$.

Breadth First Search makes a use of two additional ideas. First, throughout the algorithm we keep an ordered list of nodes which we have already visited and, if we encounter a node which is not part of that list, we add the node to the ordered list. Second, we make use of a data structure called a queue which is also an ordered list, but more dynamic - we are able to not only add nodes to the list but also delete nodes. In a queue, we are only able to delete nodes from the beginning. This scheme is known as first-in-first-out as the nodes added first are the first to be deleted. The following is the Breadth First Search algorithm explicitly. In this paper, for purposes of being precise, we
use pseudocode which closely resembles Java. We use Java in this paper because its Object-Orient Programming aspect allows us to work with objects such as Trees and Graphs directly without having to implement less abstract code on matrices and arrays which underline objects such as Trees and Graphs. The following is a typical code for the Breadth First Search algorithm.

**Code 1: Breadth First Search algorithm**

```java
BFS(Node node) {
    if (node.index != null) {
        return; //Already traversed.
    }
    int index = 0;
    Queue queue = new Queue(); //New empty (local) Queue object.
    node.index = 0;
    V.add(node); //Add reference to 'node' in (global) ordered list of Nodes.
    index++ ; //Increment index.
    queue.enqueue(node); //Add current node to the Queue object.
    while (!queue.isEmpty()) {
        //Delete first Node from queue and save as 'r'.
        Node r = queue.dequeue();
        //O(deg(r)) time where deg(r) is the number of adjacent nodes to 'r'.
        for (int i = 0; i < r.children.size(); i++) { //Loop through all of children of 'r'.
            //Save r's i-th child as 's'.
            Node s = r.children.get(i);
            //If 's' is unvisited. Operation takes linear time.
            if (V.indexOf(s) < 0) {
                s.index = index; //Mark 's' as visited.
                V.add(s); //Add reference to 's' in (global) ordered list of Nodes.
                index++;
                //We will traverse s’s children when 's' is dequeued.
                queue.enqueue(s);
            }
        }
    }
}
```

The above algorithm traverses a graph $G$ starting with some main node $u_1$ continuously adding nodes to a list $V$. At the end of the algorithm, we are left with an empty queue and a list $V$ containing nodes of the graph in the order that they were visited. BFS takes $O(n)$ time as each node in $G$ is visited exactly once (this simply says that the number of operations to linearly proportional to the
Consider the following graph $S$.

After Breadth First Search is run starting at node $a$, we have the following list $V$:

$$V = \{a, c, b, g, d, f, e\}.$$  \hfill (1)

For a given main node, $V$ is unique only up to the order in which each node’s children are listed. For example, starting with node $a$, we chose $c$ as the first of $a$’s children, followed by $b$. We could have just as well chosen to list $b$ first and then $V$ would be

$$V = \{a, b, c, d, f, g, e\}.$$  \hfill (2)

Therefore, a concise way to represent a graph is through its **adjacency matrix** which is a symmetric matrix (symmetric because the graphs we consider in this paper are undirected) with 1’s representing an edge between two nodes and 0’s otherwise. The following is an adjacency matrix for the graph $S$ (with node labels listed on the sides).

$$
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
$$  \hfill (3)

By representing a graph via its adjacency matrix we also provide an inherent listing of the children for each node in the graph. Thus, there is an exactly one list $V$ produced by Breadth First Search per adjacency matrix of a graph.
2.3 Spanning Trees

Having a connected graph \( G \) in mind, a spanning tree of \( G \) is a tree which contains all nodes of \( G \) and the only edges in the spanning tree are the edges in \( G \). In other words, to obtain a spanning tree of \( G \), delete edges of \( G \) until \( G \) has no cycles, but keep all nodes connected. For graph \( S \) from the previous section, the following are two sample spanning trees.

The above two spanning trees illustrate that a spanning tree of a graph is not unique. However, given an adjacency matrix of a graph with its inherent listing of nodes, we can deterministically produce a unique spanning tree as a result of running Breadth First Search on the adjacency matrix. Let us refer to the spanning tree produced by Breadth First Search as the BFS tree. As mentioned above, spanning trees on a graph \( G \) is a subgraph of \( G \) with some edges removed. When running Breadth First Search on a graph, the edges to be removed to produce the BFS tree are precisely the edges to nodes which we already visited. This amounts to replacing the inner-most if-statement in the Breadth First Search algorithm by the following if-else-statement. The modification is the addition of the else-statement.

**Code 2: Modification to the if-else statement of Breadth First Search algorithm**

```java
if (V.indexOf(s) < 0){
    s.index= index; //Mark 's' as visited.
    V.add(s);
    index++;
    //We will traverse s’s children when 's' is dequeued.
    queue.enqueue(s);
} else{
    //Remove edge from node 'r' to node 's'.
}
```

When running the modified Breadth First Search algorithm on graph \( S \) given the listing of the
nodes as shown in S’s adjacency matrix (matrix 3), we obtain the following spanning tree (BFS tree). Note that the listing of the nodes in the adjacency matrix (matrix 3) is simply the ordered alphabetical list of the nodes, \{a, b, c, d, e, f, g\}.

3 New Algorithm

3.1 Shortest distance

Before introducing the new algorithm, we will show several lemmas which are central to proving the correctness of the algorithm (and the idea of a unique spanning tree by using Breadth First Search just discussed) and give a better understanding for the running times. First consider some unweighted, undirected, and connected graph G with a listing V of its nodes \{u_1, u_2, \ldots, u_n\} implied by its adjacency matrix M and call the BFS tree rooted at \(u_i\) as \(T(u_i)\). Denote the shortest distance between nodes \(u, v \in G\) by \(d_G(u, v)\). Again, the shortest distance may be found by finding the shortest path (number of edges) between nodes. While the shortest distance is well-defined, the path may not be unique.

**Lemma 1**

Fix \(i \in [1, n]\). The shortest distance between \(u_i\) and any other node \(u_j\), \(d_G(u_i, u_j)\), in G is \(k\) \((d_G(u_i, u_j) = k)\) if and only if \(d_{T(u_i)}(u_i, u_j) = k\) in \(T(u_i)\). In other words, the shortest distance between \(u_i \in G\) and any other node \(u_j \in G\) is the same in \(G\) and in \(T(u_i)\), the spanning tree generated by Breadth First Search rooted at \(u_i\).

**Proof**

(\(\Leftarrow\)) With \(i\) fixed and \(T(u_i)\) generated from \(G\), suppose \(d_{T(u_i)}(u_i, u_j) = k\) for some \(j\). Since to construct \(G\) from \(T(u_i)\) amounts to inserting previously deleted edges, a shortest path (some sequence of edges) which gives \(d_{T(u_i)}(u_i, u_j)\) in \(T(u_i)\) is also contained in \(G\). Since we insert edges to construct \(G\), the shortest path from \(u_i\) to \(u_j\) is not greater than \(k\). So, \(d_G(u_i, u_j) \leq d_{T(u_i)}(u_i, u_j)\).
Pick a shortest path bearing the distance \( d_{T(u_i)}(u_i, u_j) \) for some \( j \) and let the path, without loss of generality, be \( u_i - u_a - u_b - u_j \). Suppose that as we construct \( G \) from \( T(u_i) \), we insert an edge connecting \( u_a \) and \( u_j \) (giving a decreased shortest distance). This edge would have been in \( G \) only if it was deleted during Breadth First Search only if \( u_j \) is enqueued before \( u_b \) is visited. This implies that there is a shorter distance to \( u_j \) from a node visited before \( u_b \). This is a contradiction and so the shortest distance in \( G \) is at least \( k \); so, \( d_G(u_i, u_j) \geq d_{T(u_i)}(u_i, u_j) \). Putting the two inequalities together, we see that if \( d_{T(u_i)}(u_i, u_j) = k \), then \( d_G(u_i, u_j) = k \).

\[ (\Rightarrow :) \text{ Without loss of generality, suppose that a shortest path bearing the distance } d_G(u_i, u_j) \text{ for some } j \text{ is } u_i - u_a - u_b - u_j. \text{ Observe that as we delete edges to construct } T(u_i), \text{ the shortest path from } u_i \text{ to } u_j \text{ is not decreased. So } d_G(u_i, u_j) \geq d_{T(u_i)}(u_i, u_j). \text{ Suppose that as we construct } T(u_i) \text{ we delete the edge connecting } u_a \text{ and } u_b. \text{ In Breadth First Search, we would only delete this edge if we have unqueued } u_b \text{ before visiting } u_a. \text{ This implies that there is an edge to } u_b \text{ from a node visited before } u_a. \text{ In turn, we have a contradiction as then we would have a shorter path to } u_b \text{ from } u_i. \text{ Then } d_G(u_i, u_j) \leq d_{T(u_i)}(u_i, u_j). \text{ So, if } d_G(u_i, u_j) = k, \text{ then } d_T(u_i, u_j) = k. \]

The above lemma demonstrates that the shortest distances between \( u_i \) and any other node in \( G \) is preserved when constructing \( T(u_i) \) (the BFS tree rooted at \( u_i \in G \)). In this case we can simplify our shortest distance notation to \( d(u_i, u_j) \) to mean the shortest distance from \( u_i \) to \( u_j \) in both \( G \) and \( T(u_i) \).

**Lemma 2**

Given two isomorphic graphs \( G \) and \( G' \) with an isomorphism \( f \), \( d(u, v) = d(f(u), f(v)) \) for all \( u, v \in G \). In other words, shortest distances are preserved under isomorphism.

**Proof**

Consider any shortest path \( S \) between \( u \) and \( v \) in \( G \) bearing the distance \( d(u, v) \). Since edges are preserved under \( f \) (the isomorphism of graphs), \( S \) is contained in \( G' \) and has the distance of \( d(u, v) \) between \( f(u) \) and \( f(v) \). Suppose \( S \) is not a shortest path between \( f(u) \) and \( f(v) \) and let \( T \) be the shortest path between \( f(u) \) and \( f(v) \) with distance \( d(f(u), f(v)) < d(u, v) \). Then \( T \) is preserved under the inverse mapping \( f^{-1} \) and the distance between \( u \) and \( v \) is \( d(f(u), f(v)) < d(u, v) \). This is a contradiction as \( d(u, v) \) is the shortest distance between \( u \) and \( v \). So if \( d(u, v) \) is the shortest distance between \( u, v \in G \), then \( d(u, v) \) is the shortest distance between \( f(u) \) and \( f(v) \) for all \( u, v \in G \). The Lemma is completed after making the same argument with the roles of \( G \) and \( G' \) reversed.

The Lemma above shows the well known result that the shortest distance between any two nodes is an invariant property of graphs under isomorphism.
3.2 Guessing

The brute force approach to finding an isomorphism between graphs would be to make guesses as to what nodes of the first graph should be mapped onto what nodes of the second graph. After each guess, this hypothetical algorithm would check whether or not every single edge is preserved in the forward and backward directions. Consider two graphs \(G\) and \(G'\) such that \(|G| = |G'| = n\) (if \(|G| \neq |G'|\), then these are obviously nonisomorphic as an isomorphism is a bijection of nodes). The number of conjectured isomorphisms a brute force approach will make is at most \(n!\) because for the first node in \(G\) we have \(n\) possible choices (or guesses) in \(G'\), for the second node in \(G\) we have \(n - 1\) possible guesses, etc.

In this paper we describe a way to reduce the number of guesses an algorithm has to make by employing the invariant property of shortest distance. For the next Theorem, recall that the height of a tree is a longest path from the root node and that width at level \(k\) is the number of nodes distance \(k\) away from the root node.

**Theorem 1**

Given two isomorphic graphs \(G\) and \(G'\), suppose that some \(u \in G\) is mapped to \(u' \in G'\) under some isomorphism \(f\). Then, \(T(u)\) and \(T(u')\) have the same height and the same width at every level. Moreover, if \(f(u) = u'\), then the nodes distance \(k\) away from \(u\), \(\{u_1^k, u_2^k, \ldots, u_m^k\}\) must be mapped onto the nodes distance \(k\) away from \(u'\), \(\{u_1'^k, u_2'^k, \ldots, u_m'^k\}\), i.e., \(f(\{u_1^k, u_2^k, \ldots, u_m^k\}) = \{u_1'^k, u_2'^k, \ldots, u_m'^k\}\).

**Proof**

As a direct consequence of the last Lemma, if \(d(u, v) = k\), then \(d(f(u), f(v)) = k\) for all \(u, v \in G\). Therefore, if \(f(u) = u'\), then \(f(v) = v'\) only if \(d(u, v) = d(u', v')\); so \(d(u, v) = d(u', v')\) is a necessary condition for the statement \((f(u) = u' \Rightarrow f(v) = v')\). In other words, if \(d(u, v) = k\), then \(v\) must be mapped onto some node distance \(k\) away from \(u'\). Finally, we now get the last equation in the Theorem, \(f(\{u_1^k, u_2^k, \ldots, u_m^k\}) = \{u_1'^k, u_2'^k, \ldots, u_m'^k\}\). So, \(T(u)\) and \(T(u')\) must have the same width at every level (since the isomorphism is a bijection of nodes and thus the number of nodes in the set at each level must equal the number of nodes in the image) and, in particular, the heights of the trees must coincide.

In this paper, the algorithm introduced uses the above Theorem to reduce the number of guesses we will make by checking if the heights and the widths of spanning trees match. The above Theorem helps us make the first guess as to what node in the first graph could possibly be matched with what node in the second graph. Once such a pair is chosen, the next guess is made by also checking the heights and widths of the spanning trees and whether the distance between the first pair of guesses and the second pair of guesses is preserved under the conjectured isomorphism. Suppose that our first guess gave us that \(f(u) = u'\). Whatever the second guess is, say \(f(v) = v'\) where \(v\) is a distance \(k\) from \(u\), then \(f(\{u_1^k, \ldots, v, \ldots, u_m^k\}) = \{u_1'^k, \ldots, v', \ldots, u_m'^k\}\). As we make more and more guesses,
we have to check that each new guess makes sense with respect to the previous information (about the isomorphism and what nodes are mapped to what nodes per level) obtained from each guess. In other words, given that \( f(u) = u' \) and \( f(v) = v' \), then we have to check that \( d(u, v) = d(u', v') \).

So, the gist of making a guess such as \( f(u) = u' \) is to first check that the heights of \( T(u) \) and \( T(u') \) match and then to check that the width of the trees match at each level. Unfortunately, even if the trees match in this sense, it does not necessarily mean that \( f(u) = u' \). Recall the graph \( S \), and suppose we are looking for an isomorphism between \( S \) and \( S' \) (or an automorphism in this case). The following are the BFS trees rooted at \( b \) and \( d \).

Although the trees above would have the same height and the same width at each level, there is no automorphism which maps \( b \) to \( d \) (\( d \) has a child, \( e \), which is connected only by one edge - the tree rooted at \( b \) does not share this property). Checking for the BFS trees’ heights and width is a good way to reduce the number of wrong guesses, but some guesses will still be wrong as we check necessary conditions and not sufficient conditions for determining an isomorphism. Thus we need a way to backtrack on our guesses and make new ones.

### 3.3 Backtracking

One can imagine the following situation. Take two isomorphic graphs \( G \) and \( G' \) with size \( n \) and generate \( T(u_i) \) for all \( u_i \in G \) and \( T(v_i) \) for all \( v_i \in G' \). Suppose that only two BFS trees of each graph have height \( k \), \( T(u_a) \) and \( T(u_b) \) from \( G \) and \( T(v_a) \) and \( T(v_b) \) from \( G' \). So \( f(\{u_a, u_b\}) = \{v_a, v_b\} \), but we do not know whether \( f(u_a) = v_a \) or \( f(u_a) = v_b \) where \( f \) is some isomorphism between \( G \) and \( G' \). For the sake of illustration, let \( f(u_a) = v_b \) be the correct choice by supposing that there is no isomorphism \( f' \) such that \( f'(u_a) = v_a \). Suppose that \( v_a \) is listed first and \( v_b \) is listed second in the adjacency matrix of \( G' \) and, likewise, \( u_a \) is listed first and \( u_b \) is listed second in the adjacency matrix of \( G \). Currently, the algorithm would pair \( u_a \) with \( v_a \) because the heights of the trees \( T(u_a) \) and \( T(v_b) \) match and then we move on. How do we backtrack to this wrong guess and make a new one?
Let us consider graph $S$ (above) again and make two isomorphic graphs out of $S$, calling them $G_1$ and $G_2$, by attaching two different vertex listings of $S$, $V_1 = \{b, d, a, c, g, f, e\}$ for $G_1$ and $V_2 = \{d, b, a, c, g, f, e\}$ for $G_2$. As illustrated in the previous section, the BFS trees rooted at $b \in G_1$ and $d \in G_2$ have the same height and the same width at each level. So we let the algorithm take a guess that $f(b) = d$ under some isomorphism $f$ between $G_1$ and $G_2$. Then we move on to finding a match to the second node in $G_1$ which is $d$ as listed in $V_1$. As $d \in G_2$ is already matched up with $b \in G_1$, our only options left are to match $d \in G_1$ with either $b \in G_2$ or $c \in G_2$ based on the heights and widths of BFS trees. As $b$ is listed first in $V_2$, we let the algorithm take a guess that $f(d) = b$ after additionally inspecting that $f_{G_1}(b, d) = f_{G_2}(d, b)$.

The information about our proposed isomorphism $f$ obtained after the first guess is

$$f(b) = d,$$
$$f(\{d, a, c, f\}) = \{b, c, f, e\},$$
$$f(\{g, e\}) = \{a, g\}. \tag{4}$$

The latter two lines are a direct consequence of Theorem 1 (because if $f(u) = u'$, then the nodes distance $k$ away from $u$, $\{u_1^k, u_2^k, \ldots, u_m^k\}$ must be mapped onto the nodes distance $k$ away from $u'$, $\{u_1'^k, u_2'^k, \ldots, u_m'^k\}$, i.e., $f(\{u_1^k, u_2^k, \ldots, u_m^k\}) = \{u_1'^k, u_2'^k, \ldots, u_m'^k\}$). From this information, we see that $f_{G_1}(b, d) = f_{G_2}(d, b)$. This allowed us to make the second guess, $f(d) = b$. The immediate information about our proposed isomorphism $f$ is obtained with the second guess is

$$f(d) = b,$$
$$f(\{b, c, f, e\}) = \{d, a, c, f\},$$
$$f(\{a, g\}) = \{g, e\}. \tag{5}$$

After making these two guesses, we move on to finding a match to the third node listed in $V_1$ which is $a$. As $a \in G_2$ is not yet matched up and the spanning trees rooted at $a \in G_1$ and $a \in G_2$ are of the same height and width at every level, we only have to check whether or not $f_{G_1}(a, d) = f_{G_2}(a, b)$ and $f_{G_1}(a, b) = f_{G_2}(a, d)$. Note that $f_{G_1}(a, d) \neq f_{G_2}(a, b)$ and so $f(a) \neq a$ under the assumptions that $f(b) = d$ and $f(d) = b$. We still have another candidate for the mapping.
of \( a \in G_1 \), namely \( f \in G_2 \) since it is the only other node with a spanning tree of the same height and width at every level as the spanning tree rooted at \( a \in G_1 \). We only have to check that \( f_{G_1}(a,d) = f_{G_2}(f,b) \) and \( f_{G_1}(a,b) = f_{G_2}(f,d) \). However, \( f_{G_1}(a,d) \neq f_{G_2}(f,b) \) and so \( f(a) \neq a \) under the assumptions that \( f(b) = d \) and \( f(d) = b \). In fact, we cannot find another candidate mapping for \( a \in G_1 \). This means that, together, our guesses, \( f(b) = d \) and \( f(d) = b \), are wrong assuming that \( G_1 \) and \( G_2 \) are isomorphic and we have to backtrack. In particular, given that we could not make a third guess, this means that our second guess, \( f(d) = b \), is wrong assuming our first guess is correct. So, now we are again looking for a match for the second node listed in \( V_1 \). However, we have additional information that, assuming previous guesses are correct, \( d \in G_1 \) must be matched with a node to the right of \( b \in G_2 \) (the previous wrong guess mapping for \( d \in G_1 \)) as listed in \( V_2 \). This describes how the algorithm is able to backtrack to previous guesses and attempt to find new correct guesses after reaching a stage of no more possible guesses (because of a wrong guess made sometime prior).

### 3.4 Illustration of the algorithm as a path through a tree

On the high level, we can illustrate finding of the isomorphism between graphs as following a path from the root of some tree down to a leaf of a tree where each edge represents a certain mapping of the nodes between graphs. Consider two graphs \( G \) and \( G' \) of size \( n \). Let this tree have height \( n \) (so that a path from the root to a leaf is exactly \( n \) so as to represent the number of mappings that an isomorphism between these graphs must have). For this illustration, imagine that we are looking at graphs of size 3. Consider the following tree which shows all possible path from the root of the tree down to a leaf of the tree. Notice that there are exactly \( 3! = 6 \) of these paths (which is the number of permutations of a list of size 3).

Looking at the tree above, we start at node labeled 0 which represents an initially empty conjectured mapping between graphs. We have 3 edges from node 0 each of which represents a mapping of the first node in graph \( G \) to a node in graph \( G' \). Either of these edges arrives at a node labeled 1 which represents that we have built a map of size 1 by choosing a mapping of the first node in graph \( G \) to a node in graph \( G' \). We then again make a guess arriving at a node labeled 2. At this stage we have built a map of size 2. The following tree represents a possible path that we could
take to build a full map of size 3 where the number-labeled nodes show the path.

The above path (or isomorphism) was found in linear time, but this is not the case every time since at each guess we check for necessary conditions for isomorphism and not sufficient conditions so we may make wrong guesses and we would have to backtrack. The following tree shows a more likely path where wrong guesses were made and backtracking was used to correct these choices to nevertheless arrive at a correct isomorphism.

We start at node 0, make first guess and arrive at node 1. Then we make a second guess arriving at node 2. For the sake of illustrating backtracking suppose that we cannot proceed further—we cannot find an appropriate guess after checking heights and widths. We then backtrack back to a map of size 1 (we call this move going to node 3). Again, for the sake of illustration, suppose we cannot make a new guess and we have to backtrack to the starting node 0. At this stage we know that we have tried matching node 0 to a certain node represented by the edge between node 0 and node 1/3 so we proceed to go through the edge between node 0 and node 4. We arrive at a map of size 1 after the previous 4 operations arriving at this node labeled 4. Then we go to node 5 and finish at node 6. The path from the root to the leaf is of size 3 because there are 3 edges included and this represents a full map of size 3. Each of these 3 edges represents a mapping between nodes. Collectively these mappings give us the isomorphism between the hypothetical graphs $G$ and $G'$.

### 3.5 The Algorithm

The new algorithm presented in this paper aims to return an isomorphism (a bijective function in a form of an array $a[]$ where the graph’s node $i$ at $i$ is simply mapped to the second graph’s node at $a[i]$) if one exists between two graphs $G$ and $G'$. Graphs may not be isomorphic, in which
case the algorithm returns a null array. If graphs are isomorphic, graphs may have more than one isomorphism between them, but the current algorithm only returns one such function.

Below is the implementation of the algorithm where graph objects and map objects are treated as abstract entities, the call to \texttt{BFS()} is omitted (but actually is described fully in Code 1), and, for the sake of illustration, the implementation of \texttt{checkConditions()} is omitted in this code snippet. Below is the pseudocode for the Algorithm which is implemented in the corresponding code as \texttt{Graph.areIsomorphic()}.

**Code 3: Graph Isomorphism Algorithm**

```java
Map areIsomorphic(Graph G1, Graph G2){
    int n= G1.V.length; //Number of nodes in G.
    Map map= new Map(n); //Initialize a Map object.
    if (n != G2.V.length){
        //False. Graphs are of different size!
        return null;
    }
    //Initialize 2*n spanning trees.
    AbstractTree[] trees1= new AbstractTree[n];
    AbstractTree[] trees2= new AbstractTree[n];
    //Run Breadth First Search starting from each node in both graphs.
    for (int i= 0; i < n; i++){
        trees1[i]= Graph.BFS(G1, i);
        trees2[i]= Graph.BFS(G2, i);
    }
    boolean[] matched= new boolean[n]; //Keep track of matched nodes in G2.
    int mismatched= -1;
    for (int i= 0; i < n; i++){
        int length= map.length;
        for (int j= 0; j < n; j++){
            if (matched[j] != true && j > mismatched){
                boolean match= true;
                match = checkConditions(map, trees1[i], trees2[j], match);
                if (match == true){
                    map.add(length, i, j); //Add key-value pair (i,j) to map.
                    matched[j]= true; //Node j in G2 is now matched.
                    mismatched= -1;
                    //Break at first match (s.t. j-th index in G2 is larger than ’mismatched’).
                }
            }
        }
    }
    return new Map(mismatched); //Return mismatching index.
}
```

17
if (map.length == length){ //Check if new key-value pair was not added.
    if (i-1 < 0){ //If true, we cannot find a match for the very first node in G1.
        //False. Graphs are non-isomorphic!
        return null;
    }
    //Update 'mismatched' because last key-value pair is wrong.
    mismatched= map.getValue(i-1);
    matched[mismatched]= false; //Update information about matched node in G2.
    map.pop(); //Remove the last key-value pair from map.
    //On next iteration, we will try to match i-th node in G1 with new node in G2.
    i= map.length-1;
}
//True. Graphs are isomorphic!
return map;

For the sake of completeness, below is the method Graph.checkConditions() omitted in the above code snippet.

Code 4: Method for checking necessary isomorphism conditions

boolean checkConditions(Map map, AbstractTree tree1, AbstractTree tree2, boolean match) {

    if (tree1.height != tree2.height){ //Fastest condition to check for non-isomorphism.
        match= false;
    }
    else{
        for (int k= 0; k < tree1.height; k++){ //Number of nodes a distance k away must be preserved.
            if (tree1.width[k] != tree2.width[k]){ //Number of nodes a distance k away must be preserved.
                match= false;
                break;
            }
        }
    }
}
if (match == true){
    for (int k = 0; k < map.length; k++){
        int key = map.getKey(k);
        int value = map.getValue(k);
        int keyLevel = tree1.getLevel(key);
        int valueLevel = tree2.getLevel(value);
        if (keyLevel != valueLevel){ //Check to see if shortest distance is preserved.
            match = false;
            break;
        }
    }
}
return match;

Code 3 presents the main logic of the algorithm. We want to draw the reader’s attention that the building of the conjectured isomorphism is done within the nested for-loop. However, the statement 
\[ i = map.length - 1 \]
which is exactly the idea of backtracking makes the runtime of the algorithm possibly larger than \( n^2 \) which is what one would initially expect from a nested for-loop.

We present a part of the proof of correctness of the algorithm via an additional Theorem aimed at showing the validity of the returned map. Before continuing, recall that the Map object which is used by the algorithm is no more than an array \( a[] \) which maps the \( i \)th node in graph \( G \) to \( a[i] \)th node in graph \( G' \).

**Theorem 2**
If a non-null \( map \) is returned by the Algorithm, then \( map \) represents an isomorphism between \( G \) and \( G' \), the inputs of the Algorithm.

**Proof**
If a non-null \( map \) is returned, then, evident from the implementation of the Algorithm, the nested for-loop finished execution. This implies that the variable \( i \) has been iterated until the constant \( i < n \) meaning that \( map \) is of length \( n \). Moreover, since we keep track of all matched nodes by the boolean array \( matched \), each of the nodes in graph \( G' \) is matched only once. This implies that \( map \) is at least a bijection of the nodes of graphs \( G \) and \( G' \). We have yet to show that this particular bijection of the nodes is an isomorphism of the graphs, i.e., we need to show that edges are preserved under this bijection. With the addition of the last key-value pair to the conjectured isomorphism comes the method call to \( checkConditions() \). Other than checking for the heights and widths of
the spanning trees, the method also checks whether or not the shortest distance between nodes is preserved. In particular, since we are at the stage of adding the last key-value pair, we check all pairs of nodes for whether or not the conjectures isomorphism preserves the shortest distance. Moreover, this implies that we check the adjacent edges to each node as each adjacent edge represents the shortest distance to a neighboring node. Since we check all adjacent edges of all nodes, we implicitly check all edges at the addition of this last-key value pair. Therefore, if a non-null \( map \) is returned, \( map \) is not only a bijection of the nodes, but represents an isomorphism of the graphs \( G \) and \( G' \) as all edges are preserved.

3.6 Program checking

To supplement the proof of correctness in the previous section, we present a probabilistic approach to checking whether a heuristic to checking for Graph Isomorphism is correct or buggy due to Blum and Kannan [2]. The approach gives an insight into the probability that a proposed heuristic distinguishes between non-isomorphic graphs and identifies between isomorphic graphs correctly or only by chance. The following is the proposed program checking for a Graph Isomorphism heuristic due to Blum and Kannan.

Compute \( P(G,H) \). //\( P \) is the Graph Isomorphism Algorithm.
if \( P(G,H) = \text{Yes} \), then
   Use \( P \) (as it were bug-free) to search for an ‘isomorphism’ from \( G \) to \( H \).
   Check whether the resulting correspondence is an isomorphism.
   If not, return BUGGY; if yes, return CORRECT.
if \( P(G,H) = \text{No} \), then
   Do \( k \) times:
      Toss a fair coin.
      if coin = heads then
         generate a random permutation \( G' \) of \( G \).
         Compute \( P(G,G') \).
         if \( P(G,G') = \text{No} \) then return BUGGY.
      if coin = tails then
         generate a random permutation \( H' \) of \( H \).
         Compute \( P(G,H') \).
         if \( P(G,H') = \text{Yes} \) then return BUGGY.
   End-do
Return CORRECT.
End.

The above program checker looks for a contradiction if the initial result says that \( G \) and \( H \) are non-isomorphic. Blum and Kannan state that if a proposed heuristic correctly distinguishes from
two non-isomorphic graphs only by chance, then it does so with probability \(1/2^k\). The Algorithm presented in this paper easily passes the above program check on ten pairs of random isomorphic graphs of size 128 and on ten pairs of random non-isomorphic graphs of size 128 with probability of failure being \(1/2^{100}\). This test is implemented as `ProgramChecker.java` in the accompanying code.

4 Random Graphs

4.1 Introduction to Erdős-Renyi model

We present a model for generating random graphs in order to find experimental running times. Description of random graphs allows to understand properties of graphs which directly influence the run time and further identify necessary conditions of graphs under which the performance of the Algorithm is advantageous.

In 1959, P. Erdős and A. Rényi introduced a notion of a random graph [6]. Graph \(G(n, p)\) is constructed such that it contains \(n\) (uniquely labeled) nodes and each of the possible \(\binom{n}{2}\) edges has a probability \(p\) of existing. One relevant characteristic of a random graph which the authors have studied is the probability of existence of a greatest connected component.

In the following code we generate an adjacency matrix for a random graph \(G(n, p/q)\) and give each edge the probability of \(p/q\) of existence where \(p/q\) is a rational number. We allocate \(n^2\) bits where each 1 (or `true`) value between \(i\)th and \(j\)th nodes represents an edge and 0 (or `false`) value otherwise. By using `BitSet` arrays instead of boolean arrays in Java, we greatly reduce the space needed for an adjacency matrix and we are able to optimize some matrix multiplication methods. As mentioned earlier, the use the convention of putting 0’s on the diagonal. Note that the else-statement assures that the adjacency matrix is symmetric. The following Code is implemented as `AdjMatrix.makeRandom()`.

**Code 5: Generation of Erdős-Renyi random graph \(G(n,p/q)\)**

```java
BitMatrix makeRandom(int n, int p, int q){
    BitMatrix matrix= new BitMatrix(n);
    Random random= new Random();

    for (int i= 0; i < n; i++){
        for (int j= i; j < n; j++){
            if (i == j){
                matrix.setBit(i, j, false);
            } else{
                boolean flag= (random.nextInt(q) <= p-1)? true: false;
            }
        }
    }

    return matrix;
}
```
matrix.setBit(i, j, flag);
matrix.setBit(j, i, flag);
}
}
return matrix;
}

The above matrix of bits allows us to make a Graph object by iterating over each row or column, counting the number of 1’s, and adding the corresponding nodes as children for each row of column.

4.2 Generation of Permutation Matrices

For the sake of this paper and testing purposes of the code, it is also important to be able to make permutations of adjacency matrices so that we can generate isomorphic graphs and so that we can use the program checker where we also permute graphs when we look for a possible inconsistency in the results of the algorithm. The generation of permutation matrices is done fairly easily. The following code snippet starts by allocating an array \( a[] \) of length \( n \) such that \( a[i] = i \). Then, we iterate through each index in \( a[] \) and move \( a[i] \) to a randomly chosen index \( 0 \leq j < n \). The result is a uniformly chosen random vector of length \( n \). Finally, to make an actual matrix out of this, we put 1 in \( i \)th row and \( a[i] \)th column for all \( i \) until \( n - 1 \). The following Code can be found in `PermMatrix.makeRandom()`.

**Code 6: Generation of a Random Permutation Matrix**

```java
BitMatrix makeRandom(int n){
    BitMatrix matrix= new BitMatrix(n);
    Random random= new Random();

    int[] list= new int[n];
    //Make set {1, 2, 3, ...}.
    for (int i= 0; i < n; i++)
        list[i]= i;

    //Randomly permute set.
    for (int i= 0; i < n; i++)
        int index= random.nextInt(n-i)+i;
        int temp= list[index];
        list[index]= list[i];
        list[i]= temp;

    return matrix;
}
```
Finally put 1 in i-th row and list(i)-th column for all i until n.
for (int i = 0; i < n; i++) {
    matrix.setBit(i, list[i], true);
}
return matrix;

Let P be the matrix returned by the above code. Let A be an adjacency matrix. Then, to get $A'$ with rows and columns permuted, we first permute rows by multiplying $P$ by $A$ and then permute columns by multiplying the previous product by $P^T$ (where $P^T$ is the transpose of $P$). So,

$$A' = PAP^T.$$ (6)

The PermMatrix.java class includes the method for making a random permutation matrix and also to multiply matrices. The multiplication method is optimized for the above task (by taking advantage of the sparsity of permutation matrices).

### 4.3 Growth of Random Graphs

The short investigation into the growth of random graphs helps with the analysis of the running time of the Algorithm in this paper. As mentioned in the introduction to random graphs, a big concern in the study of random graphs is the existence of a giant component.

Recall that the graph $G(n, p)$ has $n$ nodes and each of the possible $\binom{n}{2}$ edges has a probability of $p$ of existing. Let $\lambda = pn$. The value of $\lambda$ can be thought of as the expected number of adjacent nodes for each node in $G(n, p)$, or, in other words, $\lambda$ is the degree of $G$. While the growth of a random graph necessitates that $n \to \infty$, we can consider at least two ways to grow a random graph, either by keeping $\lambda$ constant or not. The experimental running times discussed in the next section include both considerations of growing a random graph. To choose interesting values for $\lambda$, we turn to whether or not a giant component exists in $G(n, p)$ as studied by Durrett in [5]. In fact, due to Durrett, we have the result that, if $\lambda < 1$, then the giant component does not exist (and all components are small) and, if $\lambda > 1$, then the giant component exists. The experimental running times presented in the next section reflect the fact that the current Algorithm is sensitive to whether or not a giant component exists (in terms of the running time).
<table>
<thead>
<tr>
<th>$n$</th>
<th>number of tests</th>
<th>min. time (s)</th>
<th>max. time (s)</th>
<th>average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1000</td>
<td>$1.140 \cdot 10^{-6}$</td>
<td>$8.039 \cdot 10^{-4}$</td>
<td>$9.758 \cdot 10^{-6}$</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>$1.900 \cdot 10^{-6}$</td>
<td>$9.590 \cdot 10^{-4}$</td>
<td>$2.532 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>12</td>
<td>1000</td>
<td>$3.801 \cdot 10^{-6}$</td>
<td>$0.005153$</td>
<td>$3.061 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>$5.222 \cdot 10^{-6}$</td>
<td>$4.552$</td>
<td>$0.006362$</td>
</tr>
</tbody>
</table>

Table 1: Test on Isomorphic graphs with $\lambda = 0.5$

5 Experimental Running Times

5.1 Introduction to results

Calculating the running times analytically proved to be a very difficult task and, instead, we present experimental running times for various parameters of random graph growth. We divide the running time results into 5 separate subsections based on the value of $\lambda$ used. We use values from $\lambda = 0.5$ (this shows results for pairs of graphs which lack a giant component) to $\lambda = 2, 16, 128$ (which show results for pairs of graphs which usually have a giant component) and the last subsection does not fix $\lambda$ (the parameter is allowed increase with $n$) and instead fixes $p = 1/2$ (this shows results for pairs of graphs where the vertex degree, the number of adjacent nodes to each node, is not fixed). In each subsection we differentiate between results where isomorphic pairs of graphs and non-isomorphic pairs of graphs were tested. Each subsection discusses the appropriate values of $n$ for which the Algorithm finds an isomorphism or verifies that there is no isomorphism in a practical and reasonable amount of time. Where appropriate (based on the range of values of $n$ used), graphs are included which aim to show the running time complexity. Each test is run anywhere from a 1000 to 2 times (depending on the amount of time the tests take). If the test takes a very short amount of time, higher number of tests are used. Since there are multiple tests for each entry, we are able to additionally provide the minimum and the maximum running times in supplement of the average time. These three values give a rough feel to the spread of the running times.

5.2 Running times when $\lambda = 0.5$

Starting at $n = 4$, we find that the average time (over 1000 pairs of random graphs) to find the solution between two isomorphic graphs takes $9.758 \cdot 10^{-6}$ seconds with the minimum and maximum times of $1.140 \cdot 10^{-6}$ and $8.039 \cdot 10^{-4}$ seconds, respectively. Results, including the values for $n = 8, 12, 16$, are summarized in the following table.

The above table demonstrates that the maximum calculation time encountered over 1000 tests grows very quickly. In fact, it was found that the Algorithm presented in this paper is inefficient at finding isomorphism between graphs larger than $n = 16$. However, the table below summarizes the results for the time for the Algorithm to complete verifying that two non-isomorphic graphs (with $\lambda = 0.5$) do not have an isomorphism between them. The reason for this may be in the
Table 2: Test on Non-isomorphic graphs with $\lambda = 0.5$

<table>
<thead>
<tr>
<th>$n$</th>
<th>number of tests</th>
<th>min. time (s)</th>
<th>max. time (s)</th>
<th>average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1000</td>
<td>$1.520 \cdot 10^{-6}$</td>
<td>$9.176 \cdot 10^{-4}$</td>
<td>$2.418 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>$1.748 \cdot 10^{-5}$</td>
<td>$0.001037$</td>
<td>$2.955 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>$3.117 \cdot 10^{-5}$</td>
<td>$0.001098$</td>
<td>$4.409 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>$8.704 \cdot 10^{-5}$</td>
<td>$0.002430$</td>
<td>$1.103 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>$3.968 \cdot 10^{-4}$</td>
<td>$0.009349$</td>
<td>$4.610 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>$0.002400$</td>
<td>$0.03378$</td>
<td>$0.002685$</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>$0.01670$</td>
<td>$0.03045$</td>
<td>$0.01803$</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>$0.1227$</td>
<td>$0.2121$</td>
<td>$0.1327$</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>$0.9541$</td>
<td>$2.053$</td>
<td>$1.166$</td>
</tr>
<tr>
<td>2048</td>
<td>10</td>
<td>$11.13$</td>
<td>$54.00$</td>
<td>$26.97$</td>
</tr>
</tbody>
</table>

The sparseness of the graphs when no giant component exists. Most of the nodes are single when $\lambda = 0.5$ and so in order to find an isomorphism, the version of the Algorithm presented in this paper must map a disconnected graph onto a disconnected graph, but mapping individual single nodes is computationally expensive in this case because many possible guesses can be made. In other words, further work on this Algorithm will include finding ways of mapping connected components onto connected components between graphs. In this way the computation can work on smaller pieces of the overall graph without having to consider the whole disconnected graph at the same time.

As evident from the above table, the Algorithm is much more efficient for verifying that non-isomorphic graphs do not have an isomorphism than at finding an isomorphism between two isomorphic graphs.

5.3 Running times when $\lambda = 2$

This is the first section when we discuss the running times on graphs which usually contain a giant component ($\lambda \geq 1$). As in the case of $\lambda = 0.5$, the graphs with $\lambda = 2$ are also sparse. Therefore, we are not surprised that we find similar running times for $\lambda = 2$, both for isomorphic and non-isomorphic pairs of graphs, as for $\lambda = 0.5$. In particular, the Algorithm presented in this paper was found to be inefficient at finding isomorphism between graphs larger than $n=32$.

Although it can be argued that low values of $\lambda$ imply that there are less edges and thus less work for the Algorithm to perform, it is also the case that low values of $\lambda$ make a very sparse, or largely disconnected graphs and thus there are more compatible spanning trees that the Algorithm has to traverse. Further work of being able to subdivide the Algorithm’s computation to work on connected components should increase the performance even on largely connected graphs such as when $\lambda = 2$. 

25
Table 3: Test on Isomorphic graphs with $\lambda = 2$

<table>
<thead>
<tr>
<th>$n$</th>
<th>number of tests</th>
<th>min. time (s)</th>
<th>max. time (s)</th>
<th>average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1000</td>
<td>$1.140 \cdot 10^{-6}$</td>
<td>0.001061</td>
<td>$1.147 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>$2.661 \cdot 10^{-6}$</td>
<td>$8.697 \cdot 10^{-4}$</td>
<td>$1.851 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>$7.982 \cdot 10^{-6}$</td>
<td>0.002279</td>
<td>$2.820 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>$2.547 \cdot 10^{-5}$</td>
<td>0.003357</td>
<td>$6.840 \cdot 10^{-5}$</td>
</tr>
</tbody>
</table>

Table 4: Test on Non-isomorphic graphs with $\lambda = 2$

<table>
<thead>
<tr>
<th>$n$</th>
<th>number of tests</th>
<th>min. time (s)</th>
<th>max. time (s)</th>
<th>average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1000</td>
<td>$1.521 \cdot 10^{-6}$</td>
<td>$8.739 \cdot 10^{-4}$</td>
<td>$2.481 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>$1.749 \cdot 10^{-5}$</td>
<td>$8.997 \cdot 10^{-4}$</td>
<td>$2.984 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>$3.079 \cdot 10^{-5}$</td>
<td>0.001155</td>
<td>$4.679 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>$8.704 \cdot 10^{-5}$</td>
<td>0.002439</td>
<td>$1.087 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>$3.967 \cdot 10^{-4}$</td>
<td>0.008682</td>
<td>$4.599 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>0.002396</td>
<td>0.03123</td>
<td>0.002670</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>0.01673</td>
<td>0.03923</td>
<td>0.01804</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>0.1217</td>
<td>0.2001</td>
<td>0.1327</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>0.9451</td>
<td>2.304</td>
<td>1.230</td>
</tr>
<tr>
<td>2048</td>
<td>10</td>
<td>11.27</td>
<td>57.52</td>
<td>28.84</td>
</tr>
</tbody>
</table>

5.4 Running times when $\lambda = 16$

This is the first section which presents the results for pairs of graphs which all have a giant component as the results are very different from the previous two sections. In particular, we found that the Algorithm is now efficient for finding an isomorphism between isomorphic graphs of size up to $n = 2048$ (the Algorithm performed poorly on sizes $n = 16$ and $n = 32$ in the previous two sections). Moreover, we can see that there is less of a range in the difference between the minimum and maximum running times of the Algorithm.

5.5 Running times when $\lambda = 128$

Like the previous section of the running times for $\lambda = 16$, this section presents the running times for another value, $\lambda = 128$, when there exists a giant component in the graph. This section shows similar running times. For example, the maximum time for verifying non-isomorphism on graphs of size 2048 is 52.86 seconds for $\lambda = 16$ and 52.34 seconds for $\lambda = 128$.

Referring to Table 7, note that we start the table at $n = 128$ as it is the smallest value we can consider for a graph with $\lambda = 128$. In fact, when $n = 128$, the graph is expected to be totally connected. Also note that with the increase of $\lambda$, we are able now find isomorphisms between graphs of size up to $n = 512$. This is the expected trend that we have seen in the previous sections - with
<table>
<thead>
<tr>
<th>$n$</th>
<th>number of tests</th>
<th>min. time (s)</th>
<th>max. time (s)</th>
<th>average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1000</td>
<td>$1.900 \cdot 10^{-5}$</td>
<td>0.001256</td>
<td>$2.977 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>$7.678 \cdot 10^{-5}$</td>
<td>0.002174</td>
<td>$1.062 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>$3.079 \cdot 10^{-4}$</td>
<td>0.006117</td>
<td>$3.630 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>0.001189</td>
<td>0.01289</td>
<td>0.001315</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>0.004879</td>
<td>0.02032</td>
<td>0.005184</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>0.02064</td>
<td>0.04937</td>
<td>0.02149</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>0.09124</td>
<td>0.1262</td>
<td>0.09812</td>
</tr>
<tr>
<td>2048</td>
<td>10</td>
<td>0.4024</td>
<td>0.4683</td>
<td>0.4206</td>
</tr>
</tbody>
</table>

Table 5: Test on Isomorphic graphs with $\lambda = 16$

<table>
<thead>
<tr>
<th>$n$</th>
<th>number of tests</th>
<th>min. time (s)</th>
<th>max. time (s)</th>
<th>average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1000</td>
<td>$3.079 \cdot 10^{-5}$</td>
<td>0.03826</td>
<td>$1.105 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>$8.819 \cdot 10^{-5}$</td>
<td>0.09098</td>
<td>$4.605 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>$4.006 \cdot 10^{-4}$</td>
<td>0.03137</td>
<td>$0.02663$</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>0.002394</td>
<td>0.01786</td>
<td>0.01786</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>0.01678</td>
<td>0.1262</td>
<td>0.1327</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>0.1216</td>
<td>0.2589</td>
<td>1.132</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>0.9475</td>
<td>52.86</td>
<td>26.75</td>
</tr>
<tr>
<td>2048</td>
<td>10</td>
<td>10.94</td>
<td>0.001189</td>
<td>0.001315</td>
</tr>
</tbody>
</table>

Table 6: Test on Non-isomorphic graphs with $\lambda = 16$

an increase in $\lambda$, we can find isomorphism between larger graphs more and more efficiently. This is because we find less and less disconnected components in more and more connected graphs.

We do not present graphs of the running times yet because the previous sections did not present us with enough data to be able to visually analyze the running times. The next section discusses a different scheme for growing a graph. We do not fix $\lambda$ and instead fix only $p = 1/2$ while $n \to \infty$.

5.6 Running times when $\lambda$ unbounded and $p = 1/2$

In this section we present results for a different way of growing a random graph. Instead of keeping the expected number of adjacent nodes fixed as $n \to \infty$, we fix only the probability of the existence

<table>
<thead>
<tr>
<th>$n$</th>
<th>number of tests</th>
<th>min. time (s)</th>
<th>max. time (s)</th>
<th>average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10</td>
<td>0.004080</td>
<td>0.01797</td>
<td>0.005729</td>
</tr>
<tr>
<td>256</td>
<td>10</td>
<td>0.01738</td>
<td>0.03467</td>
<td>0.02024</td>
</tr>
<tr>
<td>512</td>
<td>10</td>
<td>0.07044</td>
<td>0.2576</td>
<td>0.1096</td>
</tr>
</tbody>
</table>

Table 7: Test on Isomorphic graphs with $\lambda = 128$
of each edge, \( p \). In particular, we let \( p = 1/2 \) as \( n \to \infty \). As \( \lambda \) grows quickly, we would expect the running times to resemble the running times for large values of \( \lambda \) such as of \( \lambda = 128 \) in the previous section. In fact, this is so. For example, consider that average time for finding an isomorphism is 0.1411 seconds for \( \lambda \) unbounded (from Table 9) and, similarly, 0.1096 seconds for \( \lambda = 128 \) (from Table 7).

The case of \( \lambda \) unbounded presents a good set of parameters for the analysis of the running time as we can test more values of \( n \) (we had to start at \( n = 128 \) for \( \lambda = 128 \), but can start at \( n = 4 \) for \( \lambda \) unbounded). Each of the two tables below, one for the test of isomorphic pairs of graphs and one for the test of nonisomorphic pairs of graphs, is accompanied by a log-log plot of the running times. Viewing the running times via a log-log plot allows us to quickly discern whether or not the running time complexity should be expected to be exponential or polynomial as well as to get a general feel for the running time behavior of the Algorithm. On a log-log scale, if we see a straight line on a log-log plot, then we should explain a polynomial behavior and, if we see a polynomial curve, then we should expect an exponential behavior.

Graph 1 shows that our running time behavior is far quicker than how a naive algorithm would perform (\( n! \) running time of a naive algorithm would have a sharp concave up property even on a log-log scale graph). In fact, our running time behavior only has slight curve up and closer resembles

<table>
<thead>
<tr>
<th>( n )</th>
<th>number of tests</th>
<th>min. time (s)</th>
<th>max. time (s)</th>
<th>average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10</td>
<td>0.002688</td>
<td>0.03476</td>
<td>0.006218</td>
</tr>
<tr>
<td>256</td>
<td>10</td>
<td>0.01782</td>
<td>0.03049</td>
<td>0.02003</td>
</tr>
<tr>
<td>512</td>
<td>10</td>
<td>0.1232</td>
<td>0.1747</td>
<td>0.1360</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>0.9314</td>
<td>2.111</td>
<td>1.118</td>
</tr>
<tr>
<td>2048</td>
<td>10</td>
<td>10.82</td>
<td>52.34</td>
<td>27.35</td>
</tr>
</tbody>
</table>

Table 8: Test on Non-isomorphic graphs with \( \lambda = 128 \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>number of tests</th>
<th>min. time (s)</th>
<th>max. time (s)</th>
<th>average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1000</td>
<td>1.900 \cdot 10^{-6}</td>
<td>0.001128</td>
<td>1.108 \cdot 10^{-5}</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>4.562 \cdot 10^{-6}</td>
<td>8.070 \cdot 10^{-4}</td>
<td>2.428 \cdot 10^{-5}</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>1.939 \cdot 10^{-5}</td>
<td>0.001113</td>
<td>3.463 \cdot 10^{-5}</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>7.792 \cdot 10^{-5}</td>
<td>0.006195</td>
<td>1.113 \cdot 10^{-4}</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>3.805 \cdot 10^{-4}</td>
<td>0.01017</td>
<td>4.596 \cdot 10^{-4}</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>0.002416</td>
<td>0.02264</td>
<td>0.002933</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>0.01689</td>
<td>0.03694</td>
<td>0.01799</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>0.1319</td>
<td>0.2161</td>
<td>0.1411</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>0.9381</td>
<td>2.567</td>
<td>1.130</td>
</tr>
<tr>
<td>2048</td>
<td>10</td>
<td>11.25</td>
<td>48.11</td>
<td>26.09</td>
</tr>
</tbody>
</table>

Table 9: Test on Isomorphic graphs with \( \lambda \) unbounded and \( p = 1/2 \)
Next we show the running time behavior on non-isomorphic pairs of graphs.

Unlike with lower values of $\lambda$, there exists less of a difference in all three metrics of running times. For example, compare the average running time of 1.130 seconds for isomorphic tests at $n = 1024$ to a similar value of 1.133 seconds for non-isomorphic tests at $n = 1024$. Other entries in both tables (Table 9 for isomorphic tests and Table 10 for non-isomorphic tests) give similar running times as

![Graph 1: Log-log plot of data in Table 9 (with $\lambda$ unbounded and $p = 1/2$)](image)

**Table 10: Test on Non-isomorphic graphs with $\lambda$ unbounded and $p = 1/2$**

<table>
<thead>
<tr>
<th>$n$</th>
<th>number of tests</th>
<th>min. time (s)</th>
<th>max. time (s)</th>
<th>average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1000</td>
<td>$1.520 \cdot 10^{-6}$</td>
<td>$9.271 \cdot 10^{-4}$</td>
<td>$2.415 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>$1.711 \cdot 10^{-5}$</td>
<td>$9.316 \cdot 10^{-4}$</td>
<td>$2.982 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>$3.041 \cdot 10^{-5}$</td>
<td>$0.001123$</td>
<td>$4.384 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>$8.667 \cdot 10^{-5}$</td>
<td>$0.002338$</td>
<td>$1.071 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>$4.003 \cdot 10^{-4}$</td>
<td>$0.008602$</td>
<td>$4.672 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>$0.002398$</td>
<td>$0.03036$</td>
<td>$0.002657$</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>$0.01673$</td>
<td>$0.04327$</td>
<td>$0.0800$</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>$0.1213$</td>
<td>$0.2322$</td>
<td>$0.1324$</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>$0.9638$</td>
<td>$2.035$</td>
<td>$1.133$</td>
</tr>
<tr>
<td>2048</td>
<td>10</td>
<td>$10.71$</td>
<td>$52.59$</td>
<td>$27.46$</td>
</tr>
</tbody>
</table>

a line in the larger values of $\lambda$. While additional data (and therefore longer and longer computation times) are required to better judge the running time, the presently tested values of $\lambda$ and $n$ identify that the Algorithm is efficient within some prescribed parameters.
well. The lessened discrepancy between two types of tests shows that with increasing parameters, the running times should be easier to analyze. In particular, from Graph 2 we see that there is also a slight curve for the lower values of $\lambda$ and a line for the higher values of $\lambda$.

The previous tables and graphs help to examine the behavior of the Algorithm at different parameters and identify appropriate values of the parameters where the Algorithm runs in an efficient and easily predictable amount of time.

6 Discussion

In this paper we identify a problem of judging whether or not two objects (with their information written down in an adjacency matrix) are similar or even the same in some form. This has direct applications in not only in computer science and mathematics, but only in fields such as network modeling, chemical group theory, and electronics. This paper, along with the supplementary Java code provides a full and tested implementation of the algorithm for working with isomorphisms of unweighted and undirected graphs. We start with simple ideas of showing that the shortest distance between isomorphic graphs is an invariant property and we use this idea in the Algorithm to explicitly determine the possible isomorphism between graphs. The proof of the Algorithm is clear and aided by a probabilistic program checker. Before moving onto the running time results, we described basic ideas of random graphs which greatly helped with the testing during the development process. The previous section clearly identifies the values of the parameters under which the Algorithm is tested and when the Algorithm is expected to run efficiently. In addition, the previous section gives
an insight into further possible improvements for the parameters where the Algorithm falls short.

The Algorithm presented in this paper is novel (at least to the author). It solves a usual problem of determining isomorphism of objects, but in this case the notion of isomorphism is applied to graphs. This problem is typical for group theory. In fact, previously known algorithms for solving this problem of finding isomorphisms of graphs uses advanced group theory, something which this paper avoided. This problem is especially exciting since the general case of the Algorithmic complexity of the running time of determining the isomorphism between graphs is not known. The running state of the art is described by Babai where a group theoretic approach shows an exponential time complexity to the case where $\lambda$ is not bounded [1]. A non-general case of where $\lambda$ is bounded was shown to be in the polynomial time domain by Luks [7]. Both of the previous results use a graph theoretic approach aimed at finding generators of groups, but it out of the scope of this paper.

Further work on this topic of determining the Graph Isomorphism using Spanning Trees generated by Breadth First Search should involve at some major developments. As discussed in Section 5, the current Algorithm performs poorly (but not hopelessly) on largely disconnected graphs. Further work would allow the Algorithm to work with each pair connected components of the graphs independently greatly speeding up the running time. Second, finding an analytic running time complexity would benefit both the author and the reader. Lastly, this may allow for possible incorporation of ideas borrowed from Babai or Luks to incorporate some group theoretic approach in the current Algorithm.
7 References


