



UNIVERSITY OF PRETORIA

COS 700

YEAR PROJECT

---

# Edit Distance Based Digraph Similarity

---

*Author:*

Ilicia JORDAAN  
u10491504

*Supervisor:*

Dr. Linda MARSHALL

7 November 2014

**DECLARATION OF ORIGINALITY**  
**UNIVERSITY OF PRETORIA**

The Department of ..... **Computer Science** ..... places great emphasis upon integrity and ethical conduct in the preparation of all written work submitted for academic evaluation.

While academic staff teach you about referencing techniques and how to avoid plagiarism, you too have a responsibility in this regard. If you are at any stage uncertain as to what is required, you should speak to your lecturer before any written work is submitted.

You are guilty of plagiarism if you copy something from another author's work (eg a book, an article or a website) without acknowledging the source and pass it off as your own. In effect you are stealing something that belongs to someone else. This is not only the case when you copy work word-for-word (verbatim), but also when you submit someone else's work in a slightly altered form (paraphrase) or use a line of argument without acknowledging it. You are not allowed to use work previously produced by another student. You are also not allowed to let anybody copy your work with the intention of passing it off as his/her work.

Students who commit plagiarism will not be given any credit for plagiarised work. The matter may also be referred to the Disciplinary Committee (Students) for a ruling. Plagiarism is regarded as a serious contravention of the University's rules and can lead to expulsion from the University.

The declaration which follows must accompany all written work submitted while you are a student of the Department of ..... **Computer Science** ..... No written work will be accepted unless the declaration has been completed and attached.

Full names of student: ..... **Ilicia Jordaan** .....

Student number: ..... **10491504** .....

Topic of work: ..... **COS 700 Year Project: Edit Distance Based Digraph Similarity** .....

**Declaration**

1. I understand what plagiarism is and am aware of the University's policy in this regard.
2. I declare that this ..... **dissertation** ..... (eg essay, report, project, assignment, dissertation, thesis, etc) is my own original work. Where other people's work has been used (either from a printed source, Internet or any other source), this has been properly acknowledged and referenced in accordance with departmental requirements.
3. I have not used work previously produced by another student or any other person to hand in as my own.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

**SIGNATURE** ..... **Ilicia Jordaan** .....

### **Abstract**

The use of edit distance as a means of measuring the similarity between digraphs is often not very informative. Digraph edit distance considers the number of insertions, deletions and substitutions of vertices and edges required to transform one digraph into another. The edit distance is given as a single value that represents the minimum number of required edit operations, but does not disclose which edit operations it represents. This paper proposes two digraph edit distance methods that include the preservation of the edit operations.

# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
1.1	Introduction . . . . .	8
1.2	Problem Statement and Motivation . . . . .	9
1.3	Objectives . . . . .	10
1.4	Layout . . . . .	10
<b>2</b>	<b>Levenshtein Distance</b>	<b>11</b>
2.1	Introduction . . . . .	11
2.2	Calculating the Levenshtein Distance . . . . .	11
2.3	Conclusion . . . . .	12
<b>3</b>	<b>Graph Basics</b>	<b>14</b>
3.1	Introduction . . . . .	14
3.2	Graphs Overview . . . . .	14
3.3	Directed Graphs . . . . .	15
3.3.1	Digraph Definition . . . . .	15
3.3.2	Loops . . . . .	15
3.3.3	Directed Cycles . . . . .	16
3.3.4	Directed Acyclic Graph . . . . .	16
3.3.5	Supergraphs and Subgraphs . . . . .	16
3.4	Graph Implementation Techniques . . . . .	17
3.4.1	Set of Pairs . . . . .	17
3.4.2	Adjacency Matrix . . . . .	18
3.5	Conclusion . . . . .	18
<b>4</b>	<b>Sample Digraphs</b>	<b>20</b>
4.1	Introduction . . . . .	20
4.2	Digraphs . . . . .	20
4.3	Conclusion . . . . .	20
<b>5</b>	<b>Digraph Edit Distance Algorithms</b>	<b>22</b>
5.1	Introduction . . . . .	22
5.2	Graph Encoding . . . . .	22
5.3	Levenshtein Distance Algorithm . . . . .	22
5.4	Digraph Edit Distance Algorithm Proposal . . . . .	25
5.5	Two Factor Digraph Edit Distance Algorithm Proposal . . . . .	26
5.6	Conclusion . . . . .	28

<b>6</b>	<b>Comparison Results</b>	<b>29</b>
6.1	Introduction . . . . .	29
6.2	Levenshtein Distance Results . . . . .	29
6.3	Digraph Edit Distance Results . . . . .	30
6.4	Two Factor Digraph Edit Distance Results . . . . .	35
6.5	Algorithms Output Comparisons . . . . .	39
6.6	Algorithms Execution Times . . . . .	41
6.7	Conclusion . . . . .	44
<b>7</b>	<b>Future Work and Conclusion</b>	<b>45</b>
7.1	Suggestion for future work . . . . .	45
7.2	Conclusion . . . . .	45

## List of Figures

1	The Königsberg bridge problem . . . . .	8
2	Calculating the edit distance using a matrix $M$ . . . . .	13
3	Example of a directed graph . . . . .	16
4	Calculating the edit distance using two vectors $v1$ and $v2$ . . . . .	25
5	Bubble chart of Levenshtein distance values . . . . .	30
6	Digraph edit distance radar chart . . . . .	32
7	Two factor digraph edit distance radar chart . . . . .	35
8	Digraph edit distance ratios . . . . .	40
9	Algorithm execution time in microseconds line graph . . . . .	42
10	Randomly generated digraphs . . . . .	43

## List of Tables

1	Adjacency matrix representation for a graph $G$ . . . . .	18
2	Sample digraphs . . . . .	21
3	Ordered pair and vertex string encodings for the sample digraphs. .	23
4	The Levenshtein distance values for the sample digraphs . . . . .	29
5	Digraph edit distance values for the sample digraphs . . . . .	31
6	Visual indication of digraph edit operations . . . . .	34
7	Two factor digraph edit distance results . . . . .	36
8	Visual indication of two factor digraph edit operations . . . . .	38
9	Total edit distance values for all the algorithms. . . . .	39
10	Algorithm execution time in microseconds . . . . .	41
11	Edit distance results for randomly generated digraphs. . . . .	42

## List of Algorithms

1	Levenshtein Distance . . . . .	24
2	Digraph Edit Distance . . . . .	27
3	Two Factor Digraph Edit Distance . . . . .	28



# 1 Introduction

## 1.1 Introduction

Graph theory is considered by many to have originated in 1736 when the solution to the "Königsberg Bridge Problem" was published by the Swiss mathematician Leonhard Euler [8]. In the Prussian city of Königsberg, the River Pregel flowed through the city and divided the city up in four land parts. The residents of Königsberg wondered whether it was possible to walk through the city by starting at any point, crossing each bridge exactly once and finally end up at the starting position again. Leonhard Euler abstracted this system of seven bridges and four land parts to a graph having four nodes that were connected by seven edges [6]. He then used this abstraction to prove that it was impossible to traverse each edge of the graph exactly once [18].

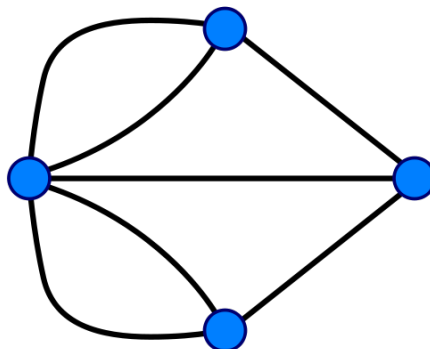


Figure 1: Königsberg bridge problem: Euler was the first to prove the impossibility of starting from any one node in this graph, traversing each edge exactly once, and ending up at the starting point again.

Since its early beginnings with the Königsberg Bridge Problem, graph theory has matured and has proven to be invaluable in a variety of fields. Graphs provide a powerful tool that can be used to model and abstract objects and relationships between objects. Examples of domains where graphs are used as models include computer networks, electric circuitry, transportation networks, biological networks, social networks, pipeline flows, computer vision, chemical compounds and many more [10, 6].

A common occurrence that is seen with the usage of graphs in these various fields is the need to somehow be able to compare different graphs to one another and measure their similarity. There are different approaches to measuring similarity between graphs and the most suitable approach for a given situation often

depends upon the application and purpose of the graphs [18]. One such an approach to determining the similarity amongst graphs is to find out how many changes one would need to make to one graph in order to transform it into another. This can be accomplished by calculating the graph edit distance, that is, the minimum number of edit operations needed to transform one graph into another [10]. The graph edit distance is a generalisation of the Levenshtein distance [13], which was originally developed to measure the similarity between two strings or sequences.

This paper looks at the Levenshtein distance or edit distance between graphs as a measure for similarity, points out the shortcomings of edit distance and proposes a modified edit distance algorithm to measure graph similarity.

## 1.2 Problem Statement and Motivation

Quantifying the similarity of graphs is an important problem with useful applications in several fields including science, networking and data analysis. One of several methods employed to measure the similarity is calculation of the graph edit distance, generally by calculating the Levenshtein distance between string encoded graphs [7]. In summary, the graph edit distance is the minimum cost of a sequence of edit operations needed to transform one graph into another. The valid edit operations are insertion, transformation and deletion of nodes and edges, and each edit operation has an associated cost.

The problem with edit distance that this paper addresses is that the output of the edit distance algorithm is a single numeric value - the minimum cost - that encompasses the cost of all the required edit operations. This value does not specify which edit operations it represents and might not always be sufficient to assess the similarity of graphs.

To demonstrate the problem, consider two different graphs, A and B. Assume that the cost of each edit operation is equal to 1 and that the minimum edit distance between A and B is 5. This edit distance value of 5 is not informative about how similar A is to B. Does 5 mean A is very similar to B or very different from B? Are parts of A similar to parts of B such that only some changes are necessary? Which edit operations does 5 represent? These are some of the questions that are left unanswered when presented with a single value for the edit distance between graphs.

### 1.3 Objectives

This paper focusses on the application of the Levenshtein distance to determine the similarity of digraphs. As part of the research presented in this paper, the author will:

- Examine the fundamental properties of graphs and look into how graphs, particularly digraphs, can be compared to one another.
- Study the Levenshtein distance, its uses and implementations.
- Consider how the Levenshtein distance can be extended to measure the similarity between digraphs.
- Identify the shortcomings of the Levenshtein distance metric as a measure for graph similarity and propose modified algorithms to address these shortcomings.
- Use the proposed algorithms to measure the similarity between sample digraphs and determine whether the proposed algorithms are comparable to the Levenshtein distance algorithm.
- Consider how the edit distance between digraphs can be represented visually.

### 1.4 Layout

The document is structured in terms of 7 chapters, each covering a different aspect of the project. Chapter 1, which is this chapter, introduces the topic of the paper and describes the motivation and problem statement addressed in this paper. Chapters 2 and 3 present the literature review of the graph theory and edit distance concepts relevant to this paper. Chapters 4 to 6 form the main body of this paper and includes the application of digraph comparisons. Chapter 4 introduces sample digraphs that are to be compared to one another by means of edit distance. Chapter 5 presents the pseudocode for the Levenshtein distance algorithm and proposes two extensions of the algorithm that can be used to measure the edit distance between digraphs. In Chapter 6, the results obtained from the algorithms for the sample digraph comparisons are presented and discussed. Finally, Chapter 7 concludes the topic discussed in this paper and offer suggestions for future work.

## 2 Levenshtein Distance

### 2.1 Introduction

The problem of quantifying and computing the similarity between strings are relevant to a variety of different fields [3] and have been applied in areas of information processing such as handwriting recognition, clone detection, signal processing, error control and gene sequence comparisons [11, 17]. One of the metrics used to quantify the similarity or difference between strings is the Levenshtein distance, also known as string edit distance [14].

This chapter introduces the Levenshtein distance as a distance metric used to measure the extent to which two strings are similar or dissimilar [2]. It provides an overview of what the Levenshtein distance is and how it can be calculated by making use of a dynamic programming method [4]. The Levenshtein distance function serves as the basis for the graph similarity comparison discussed in this paper.

In later chapters, the Levenshtein distance is used to calculate the distance between various string encoded graphs. It is also expanded upon to provide a modified edit distance algorithm that provides more edifying information than a single distance value.

### 2.2 Calculating the Levenshtein Distance

Edit distance or Levenshtein distance is a method that can be used to compare two potentially different strings to one another. The edit distance between two strings A and B is defined as the minimum cost of a number of edit operations needed in order to transform A into B [16]. The allowed edit operations are insertion of a character, deletion of a character and substitution of a character in the string. Each of the edit operations have a non-negative cost associated with the operation. The minimum cost of all the operations required to transform A to B is then added to compute the edit distance value.

The edit distance between two strings can be computed by means of a dynamic programming method [4]. Given two strings A and B, let  $|a|$  and  $|b|$  denote the length of the strings A and B respectively. The edit distance between A and B can then be computed using a matrix D with  $|a| + 1$  rows and  $|b| + 1$  columns. Let  $b[j]$  denote the  $j^{th}$  character of B, and  $b[i, j]$  denote the sub-string of B from the  $i^{th}$  to the  $j^{th}$  character.  $D[i][j]$  then denotes the edit distance between  $a[1, i]$  and  $b[1, j]$ , and

1.  $D[i][0] = i$  for each  $0 \leq i \leq |a|$

2.  $D[0][j] = j$  for each  $0 \leq j \leq |b|$
3.  $D[i][j] = \min(D[i-1][j]+1, D[i][j-1]+1, D[i-1][j-1]+\delta)$  for  $0 \leq i \leq |a|$  and  $0 \leq j \leq |b|$

where  $\delta = 0$  if  $a[i] = b[j]$ , otherwise  $\delta = 1$ .  $D[|a|][|b|]$  is then the minimum edit distance between strings A and B [4].

To give an example, consider the two strings "sitting" and "standing". If the cost of each edit operation is 1, the minimum edit distance between the strings "sitting" and "standing" is 4. To calculate this edit distance value, the matrix D is first initialised by setting the first row and column values equal to their corresponding letter's position in the string. After completion of this step, the matrix will look like the one represented in Figure 2a. After the matrix initialisation, the value for each cell in the edit distance matrix is calculated either column by column or row by row, by setting the value of the cell  $D[i][j]$  to the minimum value of:

- i. The value of the cell directly above the current cell + 1.
- ii. The value of the cell directly left of the current cell + 1.
- iii. The value of the cell diagonally left and above the current cell + cost, where the cost is 0 if the row and column letters are identical or 1 if they aren't.

After calculating the values for the first and second columns, the edit distance matrix will look like the matrix represented by Figure 2b. Figure 2c represents the state of the edit distance matrix after calculating the values for the fifth column. The final matrix D for the edit distance between the two strings is given in Figure 2d. The minimum edit distance is contained in the last cell,  $D[7][8]$ , and the minimum distance path is highlighted in yellow.

## 2.3 Conclusion

This chapter introduced the Levenshtein distance concept as a metric to measure the extent of the similarity between two strings. It demonstrated how it can be computed using a dynamic programming method and provided an example of the edit distance matrix used in the method.

The next chapter reviews some of the basic graph concepts and provides a formal definition for a directed graph. Two implementation techniques for graphs are also introduced.

		s	t	a	n	d	i	n	g
	0	1	2	3	4	5	6	7	8
s	1								
i	2								
t	3								
t	4								
i	5								
n	6								
g	7								

(a) Initialisation of the edit distance matrix

		s	t	a	n	d	i	n	g
	0	1	2	3	4	5	6	7	8
s	1	0	1						
i	2	1	1						
t	3	2	1						
t	4	3	2						
i	5	4	3						
n	6	5	4						
g	7	6	5						

(b) Distance matrix after calculating values for the second column

		s	t	a	n	d	i	n	g
	0	1	2	3	4	5	6	7	8
s	1	0	1	2	3	4			
i	2	1	1	2	3	4			
t	3	2	1	2	3	4			
t	4	3	2	2	3	4			
i	5	4	3	3	3	4			
n	6	5	4	4	3	4			
g	7	6	5	5	4	4			

(c) Edit distance matrix after calculating values for the fifth column

		s	t	a	n	d	i	n	g
	0	1	2	3	4	5	6	7	8
s	1	0	1	2	3	4	5	6	7
i	2	1	1	2	3	4	4	5	6
t	3	2	1	2	3	4	5	5	6
t	4	3	2	2	3	4	5	6	6
i	5	4	3	3	3	4	4	5	6
n	6	5	4	4	3	4	5	4	5
g	7	6	5	5	4	4	5	5	4

(d) Edit distance matrix after calculating values for the fifth column

Figure 2: Calculating the edit distance using a matrix M

## 3 Graph Basics

### 3.1 Introduction

This chapter provides an overview of the fundamental concepts and terminology of graphs, in particular that of directed graphs, so that the reader is familiar with the concepts when it is discussed in later chapters. Basic graph concepts are introduced in Section 3.2 and a formal definition for directed graphs is given in Section 3.3. Finally, two different techniques for implementing digraphs are discussed in Section 3.4.

### 3.2 Graphs Overview

This section provides a basic explanation of what constitutes a graph and introduces some of the terminology used with graphs. It also gives an overview of two different graph types, namely the undirected graph and the more specialised directed graph.

Essentially a graph is simply a collection of vertices, also called nodes, and edges, also called arcs. Vertices are connected to each other by edges to indicate a relationship between the vertices. Vertices and edges can be either labeled or unlabeled. Edges connecting two vertices can additionally be either directed or undirected.

An undirected edge is an edge where the relationship between the two vertices connected by the edge is bidirectional. An undirected edge,  $e1$ , that connects two vertices A and B, indicates that the relationship exists from both A to B *and* B to A. A directed edge is an edge where the relationship between the vertices connected by the edge exists in only one direction, with one vertex specifically defined as the source vertex, and another as the destination vertex. A directed edge,  $e1$ , that connects two vertices indicates a relationship from either A to B *or* B to A, but not both.

Undirected graphs thus consist of a set of vertices and a set of undirected edges that connect the vertices. Directed graphs consist of a set of vertices and a set of directed edges connecting the vertices.

The research presented in this paper focusses on the edit distance between directed graphs, and so the next section provides a formal definition for a directed graph and introduces some additional digraph concepts relevant to this paper.

### 3.3 Directed Graphs

#### 3.3.1 Digraph Definition

As mentioned in Section 3.2, directed graphs or digraphs consist of a set of vertices and a set of edges, with each edge having a specific source and destination vertex. The definition for directed graphs as used in this paper is one adapted from Diestel [5]. It defines a digraph  $G$  as a finite set of vertices  $V$  and directed edges  $E$ . Formally,  $G = G(V, E)$  with

- i.  $V = V(G)$ , a set of elements, called vertices of  $G$
- ii.  $E = E(G)$ , a set of edge pairs of  $G$ , with each edge pair  $\varepsilon$  being an ordered pair of  $V$  such that  $\varepsilon = (v_1, v_2)$  with  $v_1 \neq v_2$  and  $v_1, v_2 \in V$
- iii. Each edge  $\varepsilon$  is defined by two mappings, namely
  - (a) source:  $E \rightarrow V$
  - (b) destination:  $E \rightarrow V$

representing the edge  $\varepsilon = (\text{source}, \text{destination})$ , showing the direction of the edge from the source to the destination.

A visual representation of a digraph  $G$  with 5 vertices and 6 edges is given below in Figure 3. The vertices are represented by the circles and the edges are the lines connecting the vertices, with the arrows indicating the direction of the edges. The textual representation of the graph is defined according to the definition above as a digraph  $G$  with

$$\begin{aligned} V &= \{a, b, c, d, e\} \\ E &= \{(a, b), (b, c), (b, d), (c, d), (c, e), (d, e)\} \end{aligned}$$

#### 3.3.2 Loops

A loop in a digraph can be defined as an edge that connects a vertex to itself, i.e. an edge  $\varepsilon = (v_i, v_i)$  [8]. The definition for a digraph given above explicitly excludes loops as it requires that  $v_1 \neq v_2$  for each edge pair  $\varepsilon = (v_1, v_2)$ .



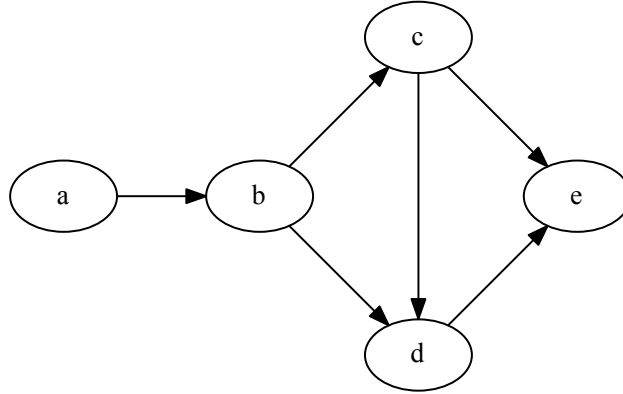


Figure 3: A directed graph with 5 nodes and 6 edges.

### 3.3.3 Directed Cycles

A directed cycle is a directed path between a sequence of vertices that starts and ends at the same vertex, such that there exists a directed edge between each two consecutive vertices [15]. A simple directed cycle is a directed cycle in which no vertices or edges are repeated aside from the start and end vertex. The length of a cycle is defined as the number of edges in the cycle.

### 3.3.4 Directed Acyclic Graph

A directed acyclic graph or DAG is a digraph without any cycles. A connected acyclic digraph is a directed tree [8].

### 3.3.5 Supergraphs and Subgraphs

A digraph  $G_1$  is a subgraph of a digraph  $G_2$  if  $V_1 \subseteq V_2$  and  $E_1 \subseteq E_2$ . If this is the case, it is denoted as  $G_1 \subseteq G_2$  and we say that  $G_2$  contains  $G_1$  [1]. Similarly,  $G_2$  is a supergraph of  $G_1$  if  $G_2$  contains  $G_1$  as a subgraph, denoted  $G_2 \supseteq G_1$ .

The next section introduces two different techniques for implementing digraphs and encoding digraphs as strings.

### 3.4 Graph Implementation Techniques

There are a number of different techniques that can be used to implement and represent digraphs. The two techniques discussed in this section were chosen primarily because they are easily converted to a string encoding, which is necessary for calculating the Levenshtein distance [7]. The first technique, discussed in Section 3.4.1, uses a set of source and destination vertex pairs to represent the digraph. The second technique, discussed in Section 3.4.2, makes use of an adjacency matrix in order to represent the vertices and edges of a graph.

#### 3.4.1 Set of Pairs

In her thesis, Marshall [12] represents a digraph as a set of triples, with each triple depicting the source and destination vertices together with the corresponding label in the form of  $(source, destination, label)$ . The definition is adapted to exclude the label element from the representation to form a set of vertex pairs. A digraph  $G$  can then be implemented as a set of vertex pairs, such that  $G = \{p_1, p_2, \dots, p_n\}$ . Each pair  $p_i = (source, destination), i \leq i \leq n$  then represents an edge of the digraph from the *source* to the *destination* vertex, with  $source \neq destination$ . This corresponds with the definition provided in Section 3.3 that defined the edges of a digraph as a set of ordered vertex pairs. The digraph shown in Figure 3 can be implemented as a set of pairs as given below.

$$G = \{(a, b), (b, c), (b, d), (c, d), (c, e), (d, e)\}$$

This representation can easily be converted into a vertex string encoding in order to be compatible with the Levenshtein distance calculation. By using only the elements of the  $(source, destination)$  pairs and removing the syntax, the corresponding vertex string encoding for the digraph is then as follows:

a b b c b d c d c e d e

Due to Levenshtein distance comparing individual characters within a string, the order of the vertices in the string encoding is important [12]. For the sake of accuracy, the edge pairs of any digraphs being compared in this manner should be sorted according to the same standards. For the purpose of this paper, the edge pairs are sorted alphabetically, first according to the source vertex and then according to the destination vertex.

### 3.4.2 Adjacency Matrix

Another possible representation for a digraph is the adjacency matrix. The adjacency matrix is an  $n \times n$  matrix depicting the edges among vertices, with  $n$  being the number of vertices [8]. Formally, the adjacency matrix of a digraph  $G$  with vertices  $v_1, v_2, \dots, v_n$  is the  $n \times n$  matrix  $M$ , with each entry  $[M]_{i,j}$  defined by

$$[M]_{i,j} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, v_i \neq v_j \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrix for the digraph shown in Figure 2 is given by the  $5 \times 5$  matrix in Table 1. The rows in the matrix represent the source vertices and the columns represent the destination vertices. The presence of a connection between vertices is indicated by a 1 and the absence of a connection is indicated by 0. Note that this is a digraph and that the direction of the connection between vertices is also encoded in the adjacency matrix, for example vertex  $a$  is connected to vertex  $b$ , but not vice versa.

		Destination vertices				
Source vertices		a	b	c	d	e
	a	0	1	0	0	0
	b	0	0	1	1	0
	c	0	0	0	1	1
	d	0	0	0	0	1
	e	0	0	0	0	0

Table 1: Adjacency matrix representation for a graph  $G$

The adjacency matrix representation can also easily be converted into a string encoding for the purposes of compatibility with the Levenshtein distance calculation. The string encoding can be generated by concatenating the 0's and 1's of each row, from top to bottom. The resulting string encoding for the adjacency matrix above is then the string 0100000110000110000100000.

## 3.5 Conclusion

This chapter introduced some of the fundamental concepts and terminology used in graph theory. It provided a formal definition for a directed graph and presented two different techniques for representing and encoding graphs. The concepts discussed in this chapter are revisited and expanded upon in later chapters when it is utilised in the graph edit distance calculations.

The next chapter provides visual representations for four sample digraphs together with their sets of vertices and edges. These sample digraphs are later compared to one another using digraph edit distance.

## 4 Sample Digraphs

### 4.1 Introduction

This chapter establishes four different sample digraphs that will be used in this paper. Each digraph is later compared to every other digraph by means of digraph edit distance in order to measure to what extent the digraphs are similar and to determine which digraphs are most similar and which ones are most dissimilar from each other.

### 4.2 Digraphs

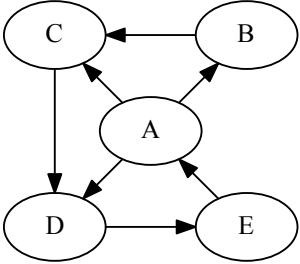
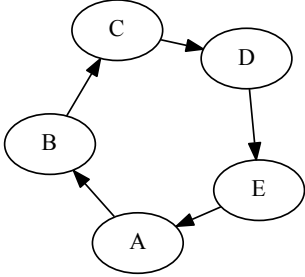
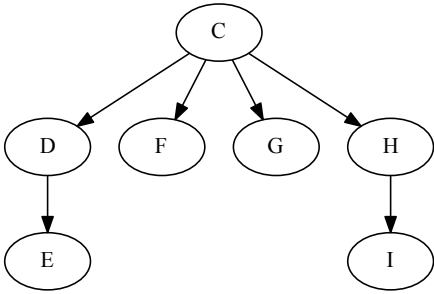
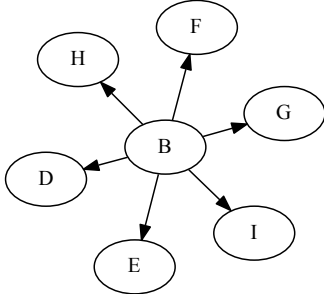
The visual representation for four sample digraphs, G1, G2, G3 and G4, are given together with their sets of vertices and edges in Table 2. The digraphs chosen are simple digraphs with no more than seven vertices and edges. These graphs will be used to test the digraph edit distance implementations and consequently the correctness of the results obtained from the edit distance comparisons should be easily verifiable by hand.

These sample digraphs were chosen from a set of some common vertices that were then connected in different ways to form different structures. Two graphs were chosen to contain cycles while the other two were chosen to be acyclic. The first digraph G1 was chosen as a starting point to represent a simple digraph with five vertices and seven edges that contains at least one directed cycle and no loops. The second digraph G2 was chosen to represent a subgraph of G1, particularly a subgraph that contains the directed cycle of the longest length. The last two digraphs were chosen to be directed acyclic graphs, essentially resulting in tree structures. G3 and G4 were chosen to share all of the same vertices except for their root nodes. They have the same number of vertices and edges, but the vertices connected by the edges differ between the digraphs. The trees formed by these graphs were chosen to have different heights, with G3 having a height of 2 and G4 a height of 1.

### 4.3 Conclusion

This chapter introduced four sample digraphs and presented their visual representations along with their sets of vertices and edges. The rationale behind the decision for choosing each digraph was also discussed. The next chapter discusses the original Levenshtein distance algorithm and proposes two extensions to it to facilitate the measurement of digraph similarity.

Table 2: Sample digraphs

<p style="text-align: center;"><b>G1</b></p>  <p style="text-align: center;"><math>V = \{A, B, C, D, E\}</math></p> <p style="text-align: center;"><math>E = \{(A,B), (A,C), (A,D), (B,C), (C,D), (D,E), (E,A)\}</math></p>	<p style="text-align: center;"><b>G2</b></p>  <p style="text-align: center;"><math>V = \{A, B, C, D, E\}</math></p> <p style="text-align: center;"><math>E = \{(A,B), (B,C), (C,D), (D,E), (E,A)\}</math></p>
<p style="text-align: center;"><b>G3</b></p>  <p style="text-align: center;"><math>V = \{C, D, E, F, G, H, I\}</math></p> <p style="text-align: center;"><math>E = \{(C, D), (C, F), (C, G), (C, H), (D, E), (H, I)\}</math></p>	<p style="text-align: center;"><b>G4</b></p>  <p style="text-align: center;"><math>V = \{B, D, E, F, G, H, I\}</math></p> <p style="text-align: center;"><math>E = \{(B,D), (B,E), (B,F), (B,G), (B,H), (B,I)\}</math></p>

## 5 Digraph Edit Distance Algorithms

### 5.1 Introduction

The digraph edit distance determines the minimum cost or minimum number of edit operations needed to transform one digraph into another. The valid edit operations for digraphs are inserts, deletes and changes to the sets of vertices and edges. For the traditional Levenshtein distance, the cost of each edit operation is typically set to 1, however these costs can be adjusted if necessary.

This chapter gives an overview of the digraph edit distance algorithms used in this paper to compare the sample digraphs. Section 5.3 describes the original Levenshtein distance algorithm implementation. Sections 5.4 and 5.4 each propose an extension of the Levenshtein distance algorithm to yield an algorithm appropriate for the measurement of digraph similarity. Before the algorithm details are discussed however, Section 5.2 reviews the digraph encodings used by the algorithms.

### 5.2 Graph Encoding

As mentioned before in Section 3.4, there are various ways to implement digraphs, two of which are by means of using an adjacency matrix or an ordered set of pairs. It was also shown in Section 3.4 how each of these implementations can be converted to a string encoding, as this is necessary to be compatible with the original Levenshtein distance calculation. Marshall [12] notes that the adjacency matrix implementation can result in a lot of computational overhead, especially when used to implement sparse digraphs, and that the execution time will increase more dramatically than other encodings as the size of the digraphs increases.

For this reason, the encoding used by the algorithms explained in the following sections are based on using a set of ordered (*source*, *destination*) pairs as described in Section 3.4.1. As the original Levenshtein distance algorithm measures the distance between two strings, the ordered pairs encoding is converted to a vertex string encoding as explained by Section 3.4.1. The ordered pairs encoding and the vertex string encoding for each of the sample digraphs are given in Table 3.

### 5.3 Levenshtein Distance Algorithm

The Levenshtein distance algorithm originally computes the edit distance between two strings and will thus require the digraphs to be encoded as vertex strings in order to compute their distances. The algorithm implementation used in this

Graph	Ordered Pairs Encoding	Vertex String Encoding
G1	$\{(A,B), (A,C), (A,D), (B,C), (C,D), (D,E), (E,A)\}$	A B A C A D B C C D D E E A
G2	$\{(A,B), (B,C), (C,D), (D,E), (E,A)\}$	A B B C C D D E E A
G3	$\{(C, D), (C, F), (C, G), (C, H), (D, E), (H, I)\}$	C D C F C G C H D E H I
G4	$\{(B,D), (B,E), (B,F), (B,G), (B,H), (B,I)\}$	B D B E B F B G B H B I

Table 3: Ordered pair and vertex string encodings for the sample digraphs.

paper is based on the dynamic programming method as explained in Section 4.2. The method takes as input two strings and then computes the edit distance between them. However, instead of constructing and using a matrix of size  $n \times m$ , it uses 2 vectors of size  $\min(n, m)$ , where  $n$  and  $m$  is the length of the two digraph strings being compared. If the length of both strings is 1000 characters, the matrix implementation will result in 1 000 000 elements being stored, where the vector implementation will store only 2000 elements. It is easy to see that this implementation is more memory efficient, and Hjelmqvist [9] claims it can be up to twice as fast as the matrix implementation when both strings compared are about 1000 characters in length.

To understand how the edit distance can be computed using two vectors or arrays, consider again the example of the strings "sitting" and "standing" compared in Section 4.2. The cost of the edit distance matrix is computed column by column, and each column can be seen as a vector. Notice from the definition of edit distance that computing the value of any cell in the edit distance matrix relies solely on the values of the cells directly above and directly left of the current cell. This means that only two vectors are necessary, one vector for the previous column and one vector for the current column.



Pseudocode for the implementation described above is given by Algorithm 1. The process of calculating the edit distance is demonstrated in Figure 4 that shows the state of the two vectors,  $v1$  and  $v2$ , at various stages of computing the edit distance cost values. Note that only the values of  $v1$  and  $v2$  need to be stored and that the values of any columns computed before  $v1$  can be discarded. The minimum edit distance cost, 4, is contained in the last index of  $v2$  in Figure 4d. Note that this state of the computation corresponds to line 20 of Algorithm 1 when  $j = 8$ , and that line 22 will swap the vectors and return the same value contained in the last index of  $v1$ , on line 24.

---

**Algorithm 1** Levenshtein Distance

---

```

1: procedure LEVENSHTEIN( $s0, s1$ )
2:    $len0 \leftarrow s0.length$ 
3:    $len1 \leftarrow s1.length$ 
4:    $v1 \leftarrow vector[len0]$ 
5:    $v2 \leftarrow vector[len0]$ 
6:   for  $i = 0$  to  $len0$  do                                ▷ Initialise vector 1
7:     set  $v1[i]$  to  $i$ 
8:   end for
9:   for  $j = 1$  to  $len1$  do                                ▷ Calculate column/vector values
10:    set  $v2[0]$  to  $j$ 
11:    for  $i = 1$  to  $len0$  do
12:      if  $s0_{i-1}$  equals  $s1_{j-1}$  then
13:         $cost = 0$                                          ▷ Characters match
14:      else
15:         $cost = 1$                                          ▷ Characters doesn't match
16:      end if
17:       $replace \leftarrow v1[i - 1] + cost$ 
18:       $insert \leftarrow v1[i] + 1$ 
19:       $delete \leftarrow v2[i - 1] + 1$ 
20:       $v2[i] \leftarrow \min(replace, insert, delete)$       ▷ Take minimum value
21:    end for
22:    swap  $v1$  and  $v2$ 
23:  end for
24:  return  $v1[len0 - 1]$                                 ▷ Return minimum edit distance cost
25: end procedure

```

---

		v1	v2							
			s	t	a	n	d	i	n	g
		0	1	2	3	4	5	6	7	8
s		1								
i		2								
t		3								
t		4								
i		5								
n		6								
g		7								

(a) Initialisation of the vectors v1 and v2

		v1	v2							
			s	t	a	n	d	i	n	g
		0	1	2	3	4	5	6	7	8
s		1	0							
i		2	1							
t		3	2							
t		4	3							
i		5	4							
n		6	5							
g		7	6							

(b) Vectors v1 and v2 when  $j = 1$

		v1	v2							
			s	t	a	n	d	i	n	g
		0	1	2	3	4	5	6	7	8
s		1	0	1						
i		2	1	1						
t		3	2	1						
t		4	3	2						
i		5	4	3						
n		6	5	4						
g		7	6	5						

(c) Vectors v1 and v2 when  $j = 2$

									v1	v2
			s	t	a	n	d	i	n	g
		0	1	2	3	4	5	6	7	8
s		1	0	1	2	3	4	5	6	7
i		2	1	1	2	3	4	4	5	6
t		3	2	1	2	3	4	5	5	6
t		4	3	2	2	3	4	5	6	6
i		5	4	3	3	3	4	4	5	6
n		6	5	4	4	3	4	5	4	5
g		7	6	5	5	4	4	5	5	4

(d) Vectors v1 and v2 when  $j = 8$

Figure 4: Calculating the edit distance using two vectors v1 and v2

## 5.4 Digraph Edit Distance Algorithm Proposal

The original Levenshtein distance method computes the edit distance between two strings as a single edit distance value that encompasses all the possible edit operations. This section proposes a modified and extended version of the edit distance algorithm discussed in the previous section.

The proposed algorithm differs from the original Levenshtein algorithm in two regards. The first way in which this algorithm is different is that it takes as input two sets of vertex pairs instead of two strings. The digraphs can thus be encoded as a set of ordered pairs to represent the edges of the digraphs. This way the digraphs are essentially compared to one another by comparing and calculating the edit distance between the edges of the two digraphs. The second way in which this algorithm differs from the Levenshtein distance algorithm is that it preserves the chosen edit operations selected at each step of the calculation. The Leven-

shstein distance method uses two vectors of integers to store the cost values. The modified digraph edit distance algorithm instead uses two vectors of custom Cost objects. Cost objects keep track of the total integer cost as well as the respective operations performed to obtain the associated cost value.

The algorithm essentially works the same in principle as the one explained in Section 5.3, however it compares edges - in the form of pairs of vertices - to one another instead of characters, and it returns a cost object instead of an integer distance value. The cost object encapsulates the edit distance cost and the edit operations performed. Algorithm 2 gives the pseudocode for the first extension to the Levenshtein distance algorithm.

## 5.5 Two Factor Digraph Edit Distance Algorithm Proposal

The second algorithm proposed is a further extension to the algorithm discussed in the previous section. This algorithm includes an additional step where the vertices of a graph are first compared using a slight variation of Algorithm 2, and then updated according to the results of the comparison. The additional operations of comparing and then updating the vertices of a digraph take place before step 2 of Algorithm 2.

Instead of taking the sets of vertex pairs as input, the algorithm receives two Digraph objects as input. The Digraph objects encapsulate the vertex pairs that represent the edges of the digraph as well the set of vertices in the digraph. The edit distance between the sets of vertices for the two digraphs is then computed in a similar manner as described in the previous section, the only difference being that sets of vertices are compared instead of sets of vertex pairs. The Cost object received by this comparison contains all the edit operations needed to transform the set of vertices of the first digraph to that of the second digraph. Using this information, the vertices in the set of vertex pairs of the first digraph can be updated accordingly before it is compared to the set of vertex pairs of the second digraph.

To better understand this implementation, consider a simple digraph D1 with vertices  $V1 = \{a, b, c\}$  and edges  $E1 = \{(a, b), (b, c), (c, a)\}$  being compared to another digraph D2 with vertices  $V2 = \{a, b, x\}$  and edges  $E2 = \{(a, b), (b, x), (x, a)\}$ . Computation of the edit distance between V1 and V2 using Algorithm 2 will return a Cost object with a total edit distance cost of 1 and a single edit operation - replacement of the vertex "c" in V1 with vertex "x" in V2. Using this information, each occurrence of vertex "c" in the set of vertex pairs E1 can be replaced with vertex "x". This will result in  $E1 = \{(a, b), (b, x), (x, a)\}$  which is the same as E2,

---

**Algorithm 2** Digraph Edit Distance

---

```
1: procedure DIGRAPHEEDITDISTANCE(Set < T > vertexPairs1, Set < T >
   vertexPairs2)
2:   len0  $\leftarrow$  nr of pairs in vertexPairs1
3:   len1  $\leftarrow$  nr of pairs in vertexPairs2
4:   v1  $\leftarrow$  vector < Cost > [len0]
5:   v2  $\leftarrow$  vector < Cost > [len0]
6:   for i = 0 to len0 do
7:     set v1[i] to Cost(i)
8:   end for
9:   for j = 1 to len1 do
10:    set v2[0] to Cost(j)
11:    for i = 1 to len0 do
12:      pair1  $\leftarrow$  vertexPairs1.pair(i - 1)
13:      pair2  $\leftarrow$  vertexPairs2.pair(j - 1)
14:      if pair1 equals pair2 then
15:        cost = Cost(0, no_op)
16:      else
17:        cost = Cost(1, op_replace(pair1, pair2))
18:      end if
19:      replace  $\leftarrow$  Cost(v1[i - 1] + cost)
20:      insert  $\leftarrow$  Cost(v1[i] + 1, op_insert(pair2))
21:      delete  $\leftarrow$  Cost(v2[i - 1] + 1, op_delete(pair1))
22:      v2[i]  $\leftarrow$  min(replace, insert, delete)
23:    end for
24:    swap v1 and v2
25:  end for
26:  return < Cost > v1[len0 - 1] ▷ Return cost object
27: end procedure
```

---

and no further edit operations will be required. Without this additional step, the edit distance cost would have been 2, as two vertex pairs would have been replaced.

It should be noted that this approach only works for replacing and deleting vertices in the set of vertex pairs, as insertion of a vertex in the set of vertices does not include information about the edges connected to the inserted vertex. Edit operations identified as vertex insertion are thus ignored and requires no change to the set of vertex pairs. Another point worth mentioning is that changes to the vertex pairs are only needed for the vertices of the first digraph, as digraph edit distance considers the operations needed in order to transform the first graph *into*

the second. Pseudocode for this implementation is given by Algorithm 3.

---

**Algorithm 3** Two Factor Digraph Edit Distance

---

```

1: procedure TWOFACTORDIGRAPHEDITDISTANCE(digraph1, digraph2)
2:   vertexPairs1  $\leftarrow$  digraph1.vertexPairs
3:   vertexPairs2  $\leftarrow$  digraph2.vertexPairs
4:   vertices1  $\leftarrow$  digraph1.vertices
5:   vertices2  $\leftarrow$  digraph2.vertices
6:   vertexDistance  $\leftarrow$  DigraphEditDistance(vertices1, vertices2)
7:   for each operation in vertexDistance.operations do
8:     if operation.type equals op_replace then
9:       applyReplacementOperation(vertexPairs1, operation)
10:    end if
11:    if operation.type equals op_delete then
12:      applyDeletionOperation(vertexPairs1, operation)
13:    end if
14:  end for
15:  continue line 2 - 25 of Algorithm 2.
16:  return  $\langle Cost \rangle v1[len0 - 1]$ 
17: end procedure

```

---

## 5.6 Conclusion

This chapter introduced the implementation of the original Levenshtein distance algorithm and proposed two digraph specific extensions to the algorithm that can be employed to calculate the edit distance between two digraphs. The first extension includes the preservation of the edit operations that constitutes the minimum edit distance value. The second extension builds on the first extension by calculating the edit distance between the vertices of the digraphs and updating the pairs of vertices accordingly before proceeding with the rest of the algorithm steps. The next chapter makes use of the algorithms discussed in this chapter to compare the sample digraphs to one another as a means of determining their similarity.

## 6 Comparison Results

### 6.1 Introduction

This chapter presents the comparison results of the sample digraphs for each of the algorithms discussed. The results for each algorithm is first discussed individually in Sections 6.2 - 6.4 before being compared to one another in Section 6.5. The average execution time for each algorithm when comparing digraphs of various sizes is presented in Section 6.6.

### 6.2 Levenshtein Distance Results

In this section, the string encoding for each of the sample digraphs is compared to that of the other sample digraphs using the original Levenshtein distance algorithm discussed in Section 5.3. Table 4 displays the minimum edit distance value or the minimum number of edit operations between each of the digraphs.

Graph Comparisons	Edit Distance
G1 and G2	4
G1 and G3	10
G1 and G4	12
G2 and G1	4
G2 and G3	8
G2 and G4	10
G3 and G1	10
G3 and G2	8
G3 and G4	9
G4 and G1	12
G4 and G2	10
G4 and G3	9

Table 4: The Levenshtein distance values for the sample digraphs

A visual representation of the edit distance between the digraphs is given as a bubble chart in Figure 5. The X and the Y axis of the chart represent the four different sample digraphs, and the bubble at each intersection represents the edit distance between the intersecting digraphs. The bubbles become larger as the edit distance values increase. The absence of a bubble at an intersection denotes an edit distance value of 0. The chart can be read from either the X or Y axis and provides a visual indication of how close one digraph is to another.

Looking at the digraph G1 on the X axis, G2 appears to be more similar to G1 than the other digraphs, while G4 appears to be more dissimilar. Overall, the two digraphs that are the most similar appears to be digraphs G1 and G2, and the digraphs that are most dissimilar are digraphs G1 and G4. Another detail noticeable from the table and bubble chart is that the order of comparisons, e.g. comparing G1 to G2 as opposed to G2 to G1, doesn't matter and that the calculation yields the same edit distance value for both comparisons. This is particularly noticeable on the bubble chart when looking at the blue line moving up from the origin of the axis, as one can see that each side of the line is basically a mirror image of the other.

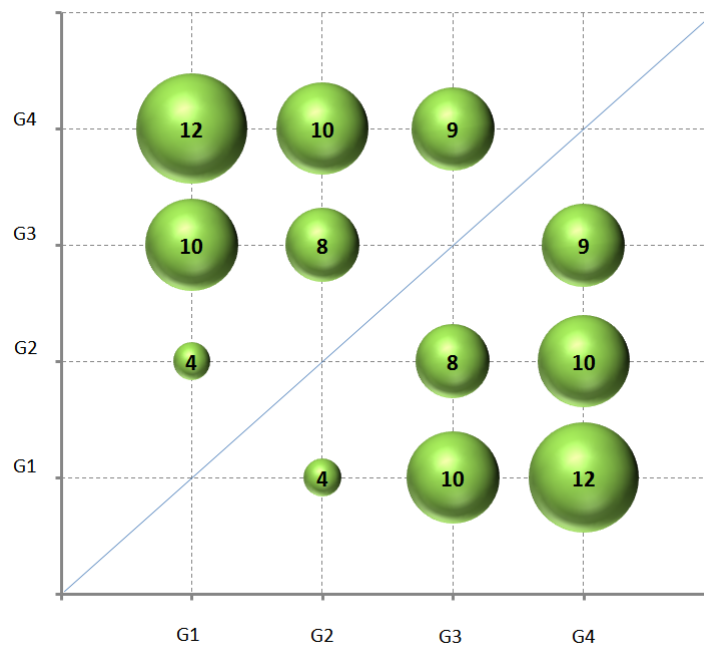


Figure 5: Bubble chart of Levenshtein distance values. The bubbles become larger as the edit distance between digraphs increase.

### 6.3 Digraph Edit Distance Results

In this section, the sample digraphs are compared using the algorithm discussed in Section 5.4. As mentioned, this algorithm keeps track of the number of inserts, deletes and substitutions required to transform the first graph into the second. For consistency purposes, the cost for each edit operation is kept to the default value of 1, however the costs associated with each operation can be changed depending

on the application. Table 5 gives the summary output of the digraph edit distance comparisons. These values are also displayed visually by means of a radar chart given in Figure 6.

Graph Comparisons	Inserts	Deletes	Substitutions	Total
G1 and G2	0	2	0	2
G1 and G3	0	1	5	6
G1 and G4	0	1	6	7
G2 and G1	2	0	0	2
G2 and G3	1	0	4	5
G2 and G4	1	0	5	6
G3 and G1	1	0	5	6
G3 and G2	0	1	4	5
G3 and G4	0	0	6	6
G4 and G1	1	0	6	7
G4 and G2	0	1	5	6
G4 and G3	0	0	6	6

Table 5: Digraph edit distance values for the sample digraphs

It was shown in Section 6.2 that the order of comparison doesn't matter for the Levenshtein distance algorithm and that the edit distance value will be the same regardless of whether G1 is compared to G2 or vice versa. This still holds true for the final cost value of the digraph edit distance, however it doesn't apply to the edit operations performed. To see why, consider the strings "abc" and "abcd". When comparing the first string to the second, the character "d" must be *inserted* into the first string in order to change it into the second string. When comparing the second string to the first, the character "d" will be *removed* from the second string in order to change it into the first string. As both of these operations have the same cost value of 1 in our implementation, the total edit distance is 1 for both sequences of comparison, however, the edit operations differ - one order of comparison results in an *insertion* operation while the other results in a *deletion* operation.

The radar chart given in Figure 6 represents the edit operations for the digraph edit distance between each of the sample digraphs. The chart includes the operations for both orders of comparison for each pair of digraphs compared, each opposite to one another on the radar chart, for example the comparison results for comparing G1 to G2 is opposite the comparison results of G2 compared to G1. From this representation, it is observed that the total digraph edit distance cost and the number of substitutions for each compared pair opposite one another is equal, but that the insert and delete operations are swapped with one another.



This is because the order of comparison can have an impact on the edit operations required to transform a digraph, as demonstrated in the paragraph above.

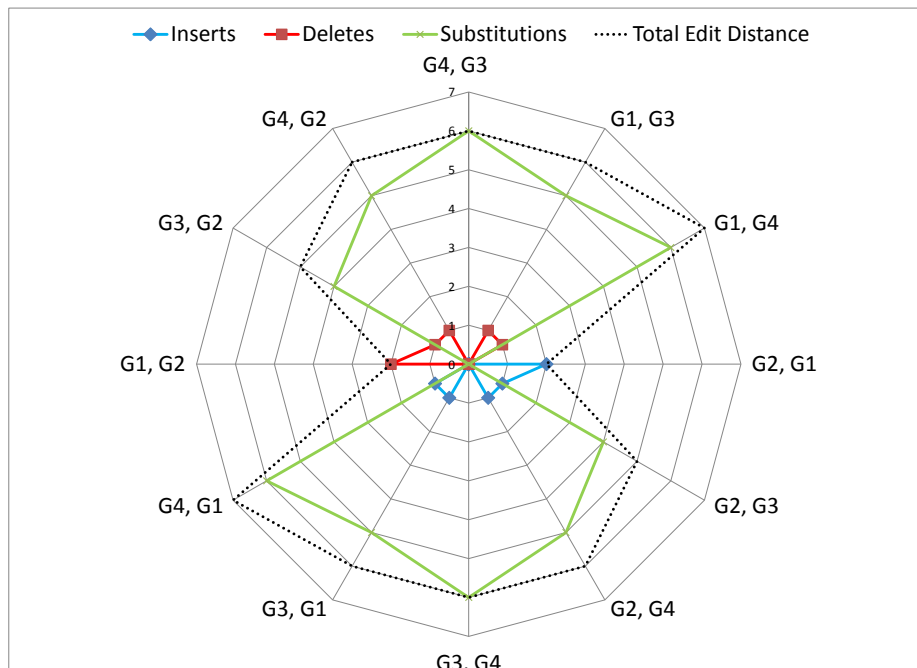


Figure 6: Radar chart showing the insertion, deletion and substitution operations for the digraph edit distance between each of the sample digraphs.

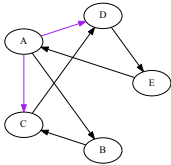
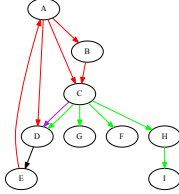
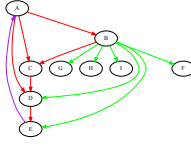
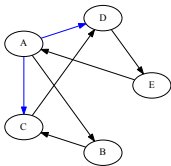
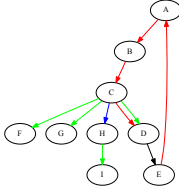
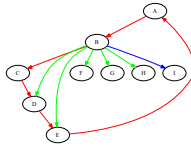
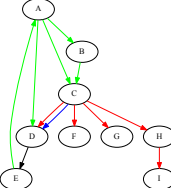
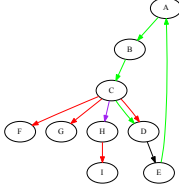
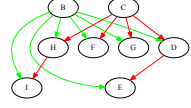
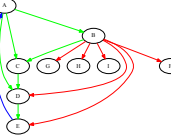
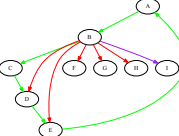
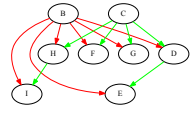
Also observable from the data in Table 5 and the radar chart in Figure 6 is that the two graphs that appear to be the most similar is G1 and G2, with a distance value of 2, while the two digraphs most different from one another appears to be G1 and G4, with a total distance value of 7. This corresponds to the output given by the Levenshtein distance method that also showed G1 and G2 to be the most similar among the digraphs and G1 and G4 to be the most different.

Since the algorithm keeps track of the edit operations performed to obtain the edit distance values, the edit operations can be displayed visually on the digraphs themselves, by using different colors to indicate edge insertions, deletions and substitutions. Table 6 provides the visual representation of the digraph edit operations. A purple line indicates that the particular edge was deleted as part of an edge deletion operation, and a blue line indicates that the edge was inserted as part of an edge insertion operation. Each edge substitution operation gener-

ates two lines, one red and the other green. Red lines indicate that the edge *was replaced* by another edge, and green lines indicate that the edge *replaced* another (red) edge. The edges that remain unchanged are indicated by black lines. The edit operations displayed for each digraph pair indicates the operations that are required to transform the first digraph into the second, for example as indicated by Table 6 (a), the two purple edges need to be deleted to transform G1 into G2.

The phenomenon of the insertion and deletion operations being swapped when the digraphs are compared in the opposite order is also visible on the digraphs in Table 6. For example, the blue edges (inserted) in Table 6 (e) are changed to purple edges (deleted) when the digraphs are compared in the opposite order in Table 6 (h). The edges that are replaced by substitution operations are also switched around as indicated by the red and green lines being swapped in the digraphs.

Table 6: Visual indication of digraph edit operations for each pair of the digraph comparisons.

<p>(a) G1,G2</p> 	<p>(b) G1,G3</p> 	<p>(c) G1,G4</p> 
<p>(d) G2,G1</p> 	<p>(e) G2,G3</p> 	<p>(f) G2,G4</p> 
<p>(g) G3,G1</p> 	<p>(h) G3,G2</p> 	<p>(i) G3,G4</p> 
<p>(j) G4,G1</p> 	<p>(k) G4,G2</p> 	<p>(l) G4,G3</p> 

## 6.4 Two Factor Digraph Edit Distance Results

This section presents the output of the digraph comparisons when compared using the two factor digraph edit distance algorithm. As explained in Section 5.5, the algorithm first calculates and applies the edit distance operations between the vertices of the digraphs before comparing the set of  $(source, destination)$  pairs. Table 7 summarises the output of the digraph comparisons, and Figure 7 presents a radar chart depicting the number of vertex and edge edit operations.

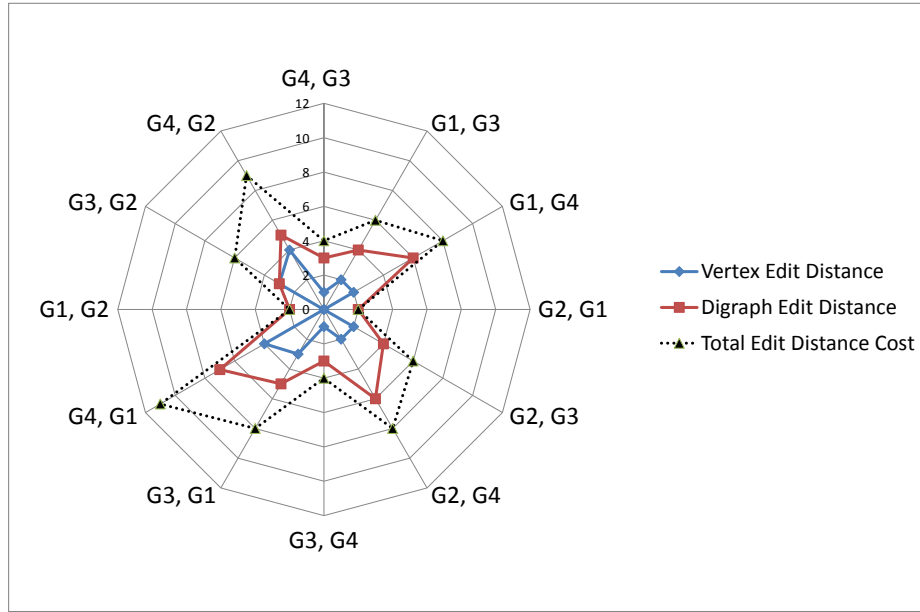


Figure 7: Radar chart showing the vertex and edge edit operations cost for the two factor digraph edit distance between each of the sample digraphs.

The radar chart given in Figure 7 is arranged in the same way as the one given in Figure 6, with each pair of digraphs placed opposite to the same pair of digraphs compared in the reverse order. From the digraph edit distance radar chart given in the previous section, it was observed that when the order of comparison for two digraphs change, the number of insertion and deletion operations are switched around, while the number of substitutions and the total edit distance values remain the same. As is evident from Figure 7, this same phenomenon is not present in the two factor digraph edit distance results. When looking at G4 compared to G1, displayed in the bottom left half of the radar chart, it is observed that the total edit distance cost is 11, while at the opposite end, with G1 compared to G4, the total edit distance cost is 8 and not 11.

Graph Comparisons	Vertex Substitutions	Vertex Deletions	Vertex Edit Distance	Inserts	Deletes	Substitutions	Digraph Edit Distance	Total Edit Distance Cost
G1 and G2	0	0	0	0	2	0	2	2
G1 and G3	0	2	2	4	0	0	4	6
G1 and G4	0	2	2	5	0	1	6	8
G2 and G1	0	0	0	2	0	0	2	2
G2 and G3	0	2	2	4	0	0	4	6
G2 and G4	0	2	2	5	0	1	6	8
G3 and G1	0	3	3	5	0	0	5	8
G3 and G2	0	3	3	3	0	0	3	6
G3 and G4	1	0	1	1	1	1	3	4
G4 and G1	0	4	4	5	0	2	7	11
G4 and G2	0	4	4	3	0	2	5	9
G4 and G3	1	0	1	1	1	1	3	4

Table 7: Two factor digraph edit distance results. The number of edit operations performed on the vertices of the graphs are combined with the digraph edit distance on the sets of vertex pairs to form the total edit distance value.

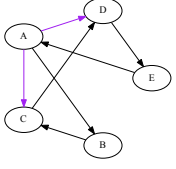
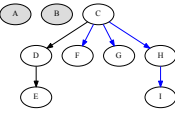
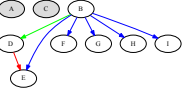
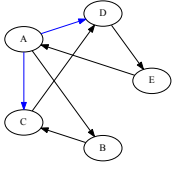
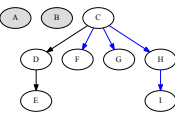
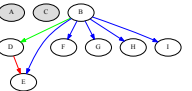
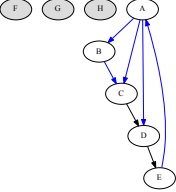
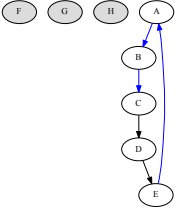
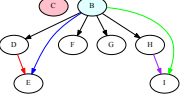
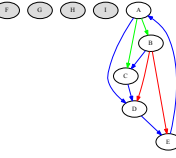
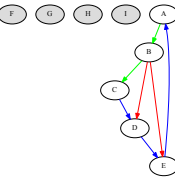
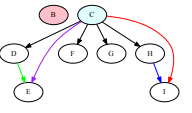
From this observation, it can be concluded that the order of comparisons of digraphs are important when compared using two factor digraph edit distance, as it can affect the edit distance values. To understand how this is possible, consider again the first part of the algorithm that calculates the edit distance between the vertices of the digraphs. The edit distance between the vertices of the first digraph and the vertices of the second digraph represent the required edit operations to transform the vertices of the first digraph *into* that of the second. If the required edit operations are all deletion of vertices, then those operations will be applied to the set of vertex pairs (that represent the edges), essentially removing all those pairs that contain the deleted vertex - either as a *source* or *destination* vertex - from the set of pairs. The digraph edit distance is then further calculated between the updated set of pairs of the first digraph and the original set of pairs of the second digraph. If these digraphs are compared in the reverse order, e.g. the second digraph is compared to the first and not vice versa, the resulting edit operations would likely change from deletion of vertices to insertion of vertices, as was noted in Section 6.3. Because insertion of vertices can not be applied to the set of vertex pairs, there will be no change to the set of vertex pairs, and the original sets of vertex pairs from both digraphs will be compared.

Taking the order of digraph comparisons into account, it is observed that the digraphs most similar to one another is G1 and G2, which corresponds with the output given by the previous algorithms. The digraphs that appear to be the most different from one another is G4 and G1, but only when the operations required to turn G4 *into* G1 are considered.

Like with the digraph edit distance operations, the two factor digraph edit distance operations can also be visualised on the digraphs themselves. The line colors used to indicate the various edit operations are the same as discussed in Section 6.3. The two factor edit distance introduces two new indicators however, one for deletion of vertices and one for substitution of vertices. Vertices that are deleted as part of a vertex deletion operation are indicated by filling the vertex with a grey color. Vertices that are replaced by another vertex are filled with red, while vertices replace another vertex are filled with green. The visualisations are represented in Table 8.

From the digraphs represented in Table 8, it is clear that the edit operations for the two factor digraph edit distance aren't necessarily merely switched around when the order of comparisons change, as was the case for the digraph edit distance results in the previous section. This is because the order of comparison has an impact on the edit distance values, as explained previously.

Table 8: Visual indication of two factor digraph edit operations for each pair of the digraph comparisons.

<p>(a) G1,G2</p> 	<p>(b) G1,G3</p> 	<p>(c) G1,G4</p> 
<p>(d) G2,G1</p> 	<p>(e) G2,G3</p> 	<p>(f) G2,G4</p> 
<p>(g) G3,G1</p> 	<p>(h) G3,G2</p> 	<p>(i) G3,G4</p> 
<p>(j) G4,G1</p> 	<p>(k) G4,G2</p> 	<p>(l) G4,G3</p> 

## 6.5 Algorithms Output Comparisons

In this section, the total edit distance values for the three algorithms are compared side by side to determine whether their results are similar. Table 9 summarises the total minimum edit distance values as calculated by each algorithm.

Graph Comparisons	Levenshtein Distance	Digraph Edit Distance	Two Factor Digraph Edit Distance
G1 and G2	4	2	2
G1 and G3	10	6	6
G1 and G4	12	7	8
G2 and G1	4	2	2
G2 and G3	8	5	6
G2 and G4	10	6	8
G3 and G1	10	6	8
G3 and G2	8	5	6
G3 and G4	9	6	4
G4 and G1	12	7	11
G4 and G2	10	6	9
G4 and G3	9	6	4

Table 9: Total edit distance values for all the algorithms.

The ratios for each digraph pair compared per algorithm can be calculated to give an indication of the similarity between the different algorithms. The ratios for the respective edit operations per algorithm can also be calculated individually. The ratios for the total edit distance cost as well as for the individual edit operations are represented visually by means of four line graphs in Figure 8.

From the line graphs it is clear that, in most of the cases, the digraph edit distance and the Levenshtein distance results are more similar than that of the two factor digraph edit distance. The number of edit operations of the digraph edit distance and Levenshtein distance are also more comparable to each other than to that of the two factor digraph edit distance. The two factor digraph edit distance is thus not always the most accurate indication of edit distance between digraphs in terms of the traditional Levenshtein distance.

From the values presented in Table 9 and the line graphs in Figure 8, it is also clear that, generally, the similarity output of each of the algorithms corresponds to that of the other algorithms, as all algorithms indicate the most similar digraphs to be G1 and G2 and the most dissimilar digraphs to be G1 and G4. The two



factor digraph edit distance can however give deviating results when the digraph pairs compared are switched around, especially when the vertex operations of the digraphs compared consist mostly of vertex insertions that change to vertex deletions when the order of comparison changes. Two factor digraph edit distance will generally only provide an advantage over the normal digraph edit distance when the digraphs being compared share a similar structure, and the labels of the vertices of the digraphs are such that the deletion or substitution of vertices will result in fewer changes needed to the set of ordered vertex pairs.

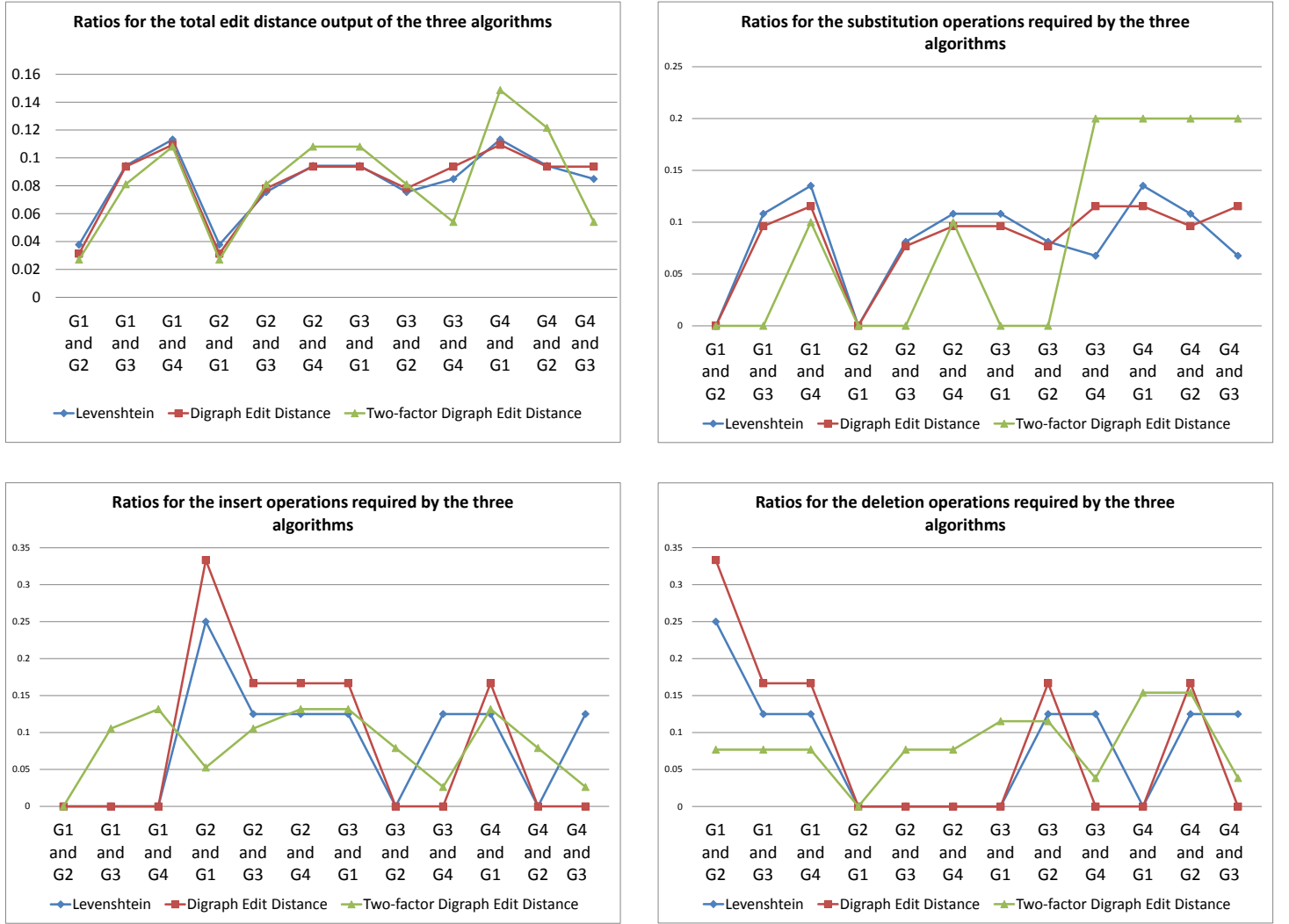


Figure 8: Ratios for the digraph edit distance values of the algorithms

## 6.6 Algorithms Execution Times

This section reviews the algorithm performance in terms of execution time taken for comparisons of digraphs with increasing numbers of vertices and edges. Table 10 gives the average execution time in microseconds for larger digraphs. The execution times given are the average execution time of 10 runs of each algorithm per digraph comparison.

The digraphs compared are randomly generated digraphs with 10, 20, 50 and 100 vertices respectively. The number of edges for each digraph is twice the number of vertices in the digraph, so the digraphs have 20, 40, 100 and 200 edges respectively. The digraph vertices were chosen at random from a pool of vertices with labels 1 - 400, and a specific amount of edges were then generated between the selected vertices. It is thus possible for the random digraphs to share no vertices or edges. The digraphs were generated so that the digraphs with fewer number of vertices are subgraphs of the digraphs with more vertices. The visual representations of the digraphs are given in Figure 10.

Number of nodes	Levenshtein Distance	Digraph Edit Distance	Two Factor Digraph Edit Distance
10	1557.2732	14622.951	28271.2048
20	6121.0078	28939.3392	47247.4098
50	7872.9472	78038.9224	107672.393
100	11317.5496	291193.4574	255518.3926

Table 10: Execution time in microseconds per algorithm for graphs with different numbers of vertices and edges.

A visual indication of the execution times per algorithm is given by the line graph in Figure 9. From the line graph it is seen that the digraph edit distance and the two factor digraph edit distance performs worse than the original Levenshtein distance. The digraph edit distance seems to perform better than the two factor digraph edit distance for digraphs with a smaller number of vertices and edges, up until about 75 vertices, where after the two factor digraph edit distance seems to perform better. This can be attributed to the deletion of vertices in the first step of the two factor digraph edit distance, essentially resulting in fewer vertex pairs that need to be compared, and consequently smaller vectors to store the edit distance values. The execution times for the proposed algorithms increase as the number of the vertices and edges increase and is proportional to the sizes of the two digraphs being compared, thus the performance for larger digraph comparisons could be less than desirable.

The results for the comparisons of the randomly generated digraphs are given in Table 11. The large edit distance values are attributed to the fact that the digraphs share no edges and very few vertices. The digraph edit distance will thus need to substitute each pair in the set of ordered vertex pairs, while the two factor digraph edit distance will additionally remove or substitute some vertices of the first digraph, resulting in additional cost.

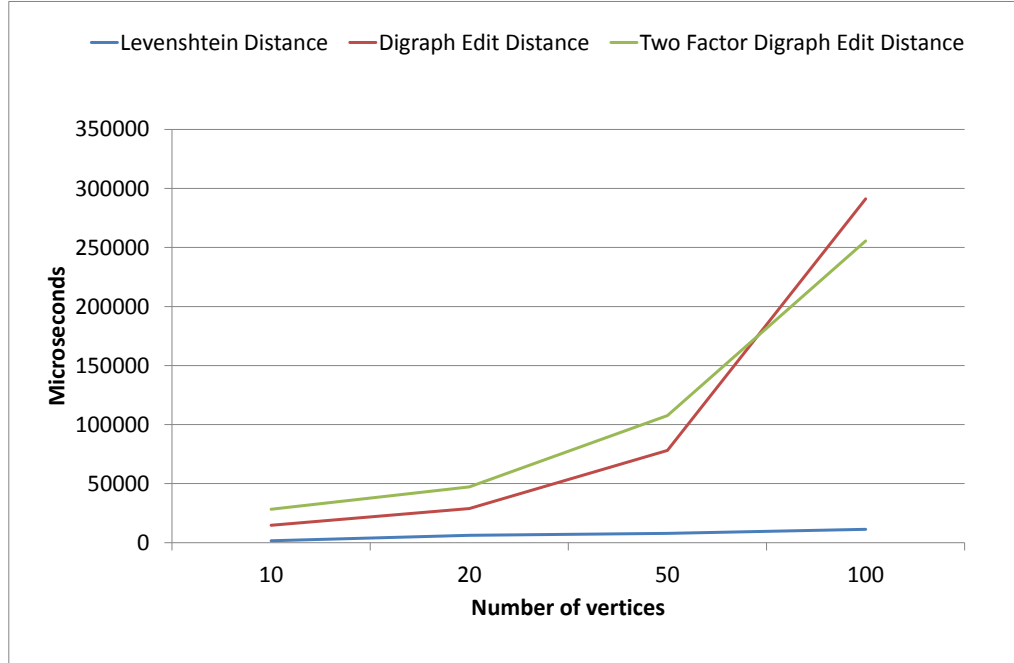


Figure 9: Line graph showing the execution time in microseconds per algorithm and number of vertices.

Digraphs Number of Nodes	Levenshtein Distance	Digraph Edit Distance	Two Factor Digraph Edit Distance
10	79	20	28
20	148	40	57
50	363	100	143
100	721	200	277

Table 11: Edit distance results for randomly generated digraphs.

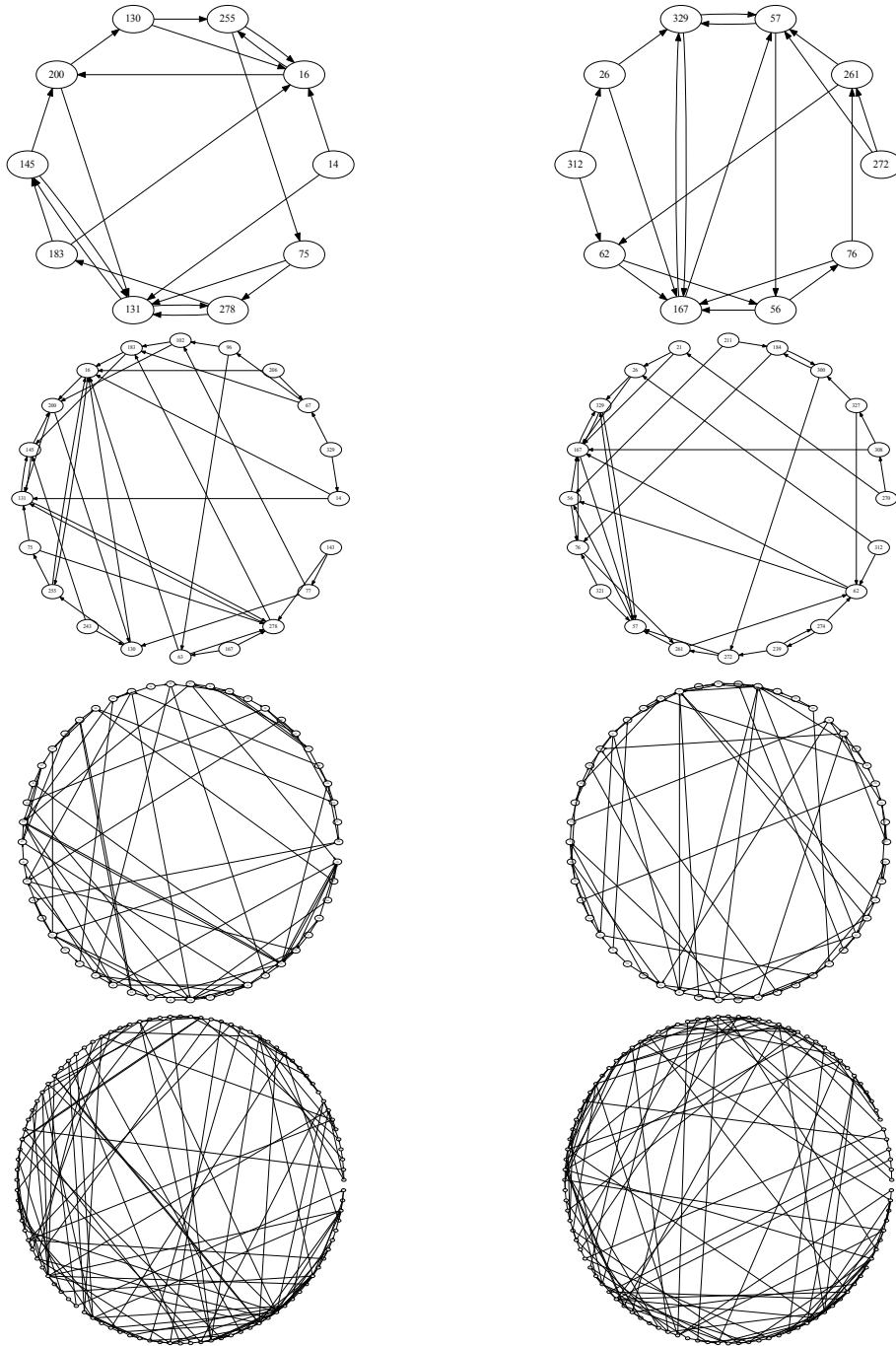


Figure 10: Randomly generated digraphs with increasing number of vertices and edges.

## 6.7 Conclusion

This section presented the comparison results of the sample digraphs for each of the algorithms discussed. The algorithms were also compared to one another and it was seen that, between the two algorithm extensions proposed, the digraph edit distance algorithm is in most cases more comparable to the traditional Levenshtein distance, while the two factor digraph edit distance algorithm is less comparable to the Levenshtein distance. All three algorithms identified the same two digraphs, G1 and G2, as the most similar among the sample digraphs, and G1 and G4 as the most dissimilar. It was shown that the order of comparison doesn't have an effect on the total edit distance values when compared using the Levenshtein and digraph edit distance algorithms, but that it can have an affect when compared using the two factor digraph edit distance algorithm. The execution times for each of the algorithms were given for randomly generated digraphs with 10, 20, 50 and 100 nodes and 20, 40, 100 and 200 edges respectively, and it was observed that the execution times for the extended algorithms are worse than the original Levenshtein algorithm.

## 7 Future Work and Conclusion

### 7.1 Suggestion for future work

The work discussed in this paper can be extended and expanded upon in several areas. Suggestions for future work include improvements of the algorithms, especially in terms of execution time, so that it can be applied to large graphs that originate from real world problems. As the algorithms discussed in this paper compared the digraphs using sets of ordered vertex pairs, it would be beneficial to also test the algorithms by comparing digraphs using sets of unordered vertices and vertex pairs, so that it can be determined whether the algorithms perform better with sets of ordered or unordered vertex pairs. Another idea that could be considered is to first extract the common subsets of vertex pairs and then compare only the remaining vertex pairs. The results of the proposed algorithms can also be compared to the results of other digraph similarity measurement techniques in order to determine the accuracy of the results.

### 7.2 Conclusion

This paper considered how the traditional Levenshtein distance method could be extended in order to measure digraph similarity. Two algorithm extensions were proposed, namely the digraph edit distance algorithm and the two factor digraph edit distance algorithm. Both proposed algorithms preserve the edit operations by means of using custom cost objects instead of integers. The two factor digraph edit distance algorithm differs from the normal digraph edit distance algorithm in that it essentially preprocesses the vertices of the first digraph before calculating the edit distance between the ordered sets of vertices of the digraphs. It was shown that two factor digraph edit distance is not always as accurate as the normal digraph edit distance, as the minimum edit distance value can vary when the digraphs are compared in the opposite order. It was also illustrated how the various vertex and edge edit operations could be displayed visually by means of coloring the vertices and edges of the digraphs, with each color depicting a specific edit operation. The execution times of the algorithms for digraphs of increasing sizes were compared, and it was found that the execution times for the proposed algorithms are worse than that of the original Levenshtein algorithm and increase as the sizes of digraphs increase. Overall, the edit distance algorithm used to measure the similarity of digraphs will be dependant upon its application. If a single distance value is desired, the traditional Levenshtein distance method can be employed. If the edit operations represented by the edit distance values are required, one of the proposed digraph edit distance algorithms can be used, with the choice of which one to use being dependant upon the digraphs and their applications.

## References

- [1] J. A. Bondy and U. S. R. Murty. *Graph theory*. Springer, 2008.
- [2] L. Cheng and H. C. Ferreira. Rate-compatible pruned convolutional codes and viterbi decoding with the levenshtein distance metric applied to channels with insertion, deletion, and substitution errors. 1:137–143, 2004.
- [3] Bhattacharya U. Chowdhury, S. D. and S. K. Parui. Online handwriting recognition using levenshtein distance metric. pages 79–83, 2013.
- [4] Li G. Feng J. Deng, D. and W. S. Li. Top-k string similarity search with edit-distance constraints. pages 925–936, 2013.
- [5] R. Diestel. *Graph Theory*. Springer-Verlag, 3 edition, 2005.
- [6] Gale. Graphs. *World of Computer Science*, 2007.
- [7] Xiao B. Tao D. Gao, X. and X. Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129, 2010.
- [8] Hirst J. L. Harris, J. M. and M. J Mossinghoff. *Combinatorics and graph theory*. Springer, 1 edition, 2000.
- [9] S. Hjelmqvist. Fast, memory efficient levenshtein algorithm - codeproject, 2012.
- [10] Parikh A. Ramdas A. Koutra, D. and J. Xiang. Algorithms for graph similarity and subgraph matching. 2011.
- [11] T. Lavoie and E. Merlo. An accurate estimation of the levenshtein distance using metric trees and manhattan distance. pages 1–7, 2012.
- [12] Linda Marshall. *A graph-based framework for comparing curricula*. PhD thesis, University of Pretoria, 2014.
- [13] Wison R. C. Myers, R. and E. R. Hancock. Bayesian graph edit distance. *IEEE Trans. Pattern Anal. Machine Intell.*, 22(6):628–635, 2000.
- [14] S. Rane and W. Sun. Privacy preserving string comparisons based on levenshtein distance. pages 1–6, 2010.
- [15] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, 2011.
- [16] Sharvit D. Klein P. Tirthapura, S. and B. B. Kimia. Indexing based on edit-distance matching of shape graphs. pages 25–36, 1998.

- [17] L. Yujian and L. Bo. A normalized levenshtein distance metric. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(6):1091–1095, 2007.
- [18] L. Zager. *Graph similarity and matching*. PhD thesis, Massachusetts Institute of Technology, 2005.