UNIVERSITY OF THE WITWATERSRAND

A DISSERTATION SUBMITTED IN THE FULFILMENT OF THE REQUIREMENT FOR THE MASTER OF SCIENCE IN THE SCHOOL OF MATHEMATICS

THE ROLE OF GRAPH THEORY IN BIG DATA ANALYTICS

Author:
Mr. Kellen Mashia

Supervisors:
Prof. Elizabeth Jonck
Prof. Terence Van Zyl

DECLARATION

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.

(Signature of candidate)

24^{th} day of July 2019 at University of the Witswatersrand

Abstract

Subgraph isomorphism detection is a computationally expensive problem which determines if there exists a subgraph $G_s = (V_s, E_s)$ in a large graph (called the data graph) denoted by $G = (V_G, E_G)$; which matches some other given smaller graph $Q = (V_Q, E_Q)$ (called the pattern graph). According to J. R. Ullmann in [19], the subgraph isomorphism problem can be solved through a brute-force enumeration procedure. In this dissertation we extensively study the Ullmann algorithm for subgraph isomorphism detection along with some of the improvements developed in [3]; and then we shall discuss the performance of the algorithm by conducting a simulation analysis for distinct, randomly generated undirected graphs. The algorithm is theoretically compared with the one proposed by L. P. Cordella in [6]. However, we only carried out an efficiency analysis on the Ullmann algorithm due to lack of large sets of experimental data. Much work has been done in this field, however, it still stands to be an open question in computer science to improve the execution run-time of the algorithm. In future research, we intend to revisit the graph and subgraph isomorphism detection problems as a possible PhD research in Computer Science. In chapter one we introduced preliminary definitions which may be of great help in understanding the main work in chapter two. In chapter two we study in detail the Ullmann algorithm for the subgraph isomorphism problem which we then implement on several examples to examine the execution run-time of the algorithm. The purpose of chapter three is to introduce another useful algorithm for subgraph isomorphism detection, called the VF2. In this chapter we study in detail the VF2 algorithm for subgraph isomorphism detection. In chapter four we give a detailed simulation analysis by implementing the Ullmann algorithm on one large data graph $G = (V_G, E_G)$ of order 30 and size 30, along with 15 randomly generated undirected graphs $Q = (V_Q, E_Q)$. In this chapter we examine the algorithms efficiency by applying it on all 15 random graphs to search for isomorphic subgraphs in the large data graph. Chapter five is the concluding chapter and chapter six entails the appendices.

Dedication

To my family; the Matlhaga family and Mashia family, this one is for you! Thank you for the love, guidance and prayers through every little step taken.

Acknowledgements

I would like to appreciate Prof Betsie Jonck and Prof Terence Van Zyl for the continued and relentless support throughout my dissertation. It has been a long way coming to the finish line, thank you so much for everything.

Contents

0.1 Basic Notation

- $Q = (V_Q, E_Q)$ is the pattern graph with vertex set V_Q and edge set E_Q.
- $G = (V_G, E_G)$ is the data graph with vertex set V_G and edge set E_G.
- $A_Q = [a_{r,c}]$ represents the adjacency matrix of the pattern graph Q.
- B_G represents the adjacency matrix of the data graph G.
- n_Q and n_G represents the number of vertices in the vertex set of the pattern graph, and the number of vertices in the vertex set of the data graph, respectively.
- (v, u) represents an edge between two given vertices v and u.
- $deg_G(u)$ represents the degree of a vertex $u \in V_G$
- $N_G(u)$ represents the open neighborhood of vertex u.
- $R \subseteq V_Q \times V_G$ represents the relation set between vertex set V_Q and vertex set V_G.
- $\delta : V_Q \mapsto V_G$ represents the bijective map, also known as a permutation between vertex sets V_Q and V_G.
- Θ represents the incidence map which assigns edges to vertices in a graph.
- $P^0 = [p^0_{r,c}]$ represents the initialized permutation matrix with entries $p^0_{r,c}$.
- χ represents the set of all possible candidate permutation matrices.
- B represents the set of all possible candidate adjacency matrices.
- i represents a state contained a state space S.
- i_t represents the target state.
- $R(i)$ represents the partial mapping solution between vertex sets V_Q and V_G.
- V_s and V_{G_s} both represent the vertex set of a subgraph.
- $F(i, v, u)$ represents the feasibility function.
- $F_{str(i,v,u)}$ is the structural feasibility function.
- $F_{sem}(i, v, u)$ is the semantic feasibility function.
- $\mathcal{T}_G(i) \subseteq V_G(i)$.
- $\mathcal{T}^{out}_Q(i)$ and $\mathcal{T}^{out}_G(i)$ represent the vertex sets containing vertices that are not included in the partial mapping solution.
- $\mathcal{T}^{in}_Q(i)$ and $\mathcal{T}^{in}_G(i)$ represent the vertex sets containing vertices which are source vertices.
- $\mathcal{M}(G, u)$ represents the predecessor set of vertex u.
- $\mathcal{C}_{pred}(i, v, u)$ represents the predecessor feasibility rule.
- $\mathcal{C}_{succ}(i, v, u)$ represents the successor feasibility rule.

- $\mathcal{S}(G, u)$ represents the successor set.
- $\mathcal{P}_Q(i)$ and $\mathcal{P}_G(i)$ represent the candidate pair sets.

Chapter 1

1.1 Introduction

The theory of graphs can arise in several diverse situations which can be of great use in solving certain graph-related application problems. Graph Theory can be effectively applied to represent search engines, DNA-sequencing, social networks, chemical structures and so forth. The purpose of this dissertation is to understand the link between Graph Theory and Big Data Analytics. Which is why we decided to study in detail, a well known open problem in Computer Science, that is the subgraph isomorphism detection problem. I am currently registered for a second masters in Computer Science and we intend to revisit the problem as a possible PhD research problem.

In this section we present the most important preliminaries, which we may find very useful in understanding the sections to follow.

A *graph* denoted by $G = (V_G, E_G)$ is a finite, nonempty set of vertices V_G and a (possibly empty) set $E_G \subset V_G \times V_G$ of (ordered or unordered) pairs of distinct vertices in the graph G. The set V_G contains vertices of a graph G, and these vertices in the graph G can be referred to as points in the graph. Furthermore, there are relationships between certain vertices in the graph G, which we refer to as the pairs of vertices in the graph G, and these shall also be called edges or links between distinct vertices in the graph G, [15].

To further understand the concept behind edges and vertices we introduce a mapping, see below

$$\Theta : E_G \mapsto V_G \times V_G,$$

which assigns an edge or a link between given vertices in the graph G. In particular, Θ maps an edge from the edge set E_G into the Cartesian product of the vertex sets $V_G \times V_G$. Here, two given vertices in V_G are assigned an edge; that is to say, the given vertices are linked to each other by the use of an edge.

Such a mapping is called an *incidence mapping*. The incidence mapping further defines two more mappings from the edge set E_G to the vertex set of the graph G, that is,

$$\alpha : E_G \mapsto V_G, \;\; \beta : E_G \mapsto V_G$$

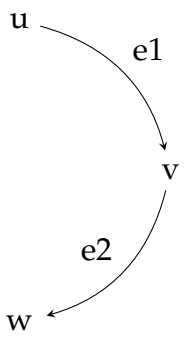


Figure 1.1: Directed Graph with 3 vertices

defined by $\Theta(e) = (\alpha(e), \beta(e))$ for any given edge $e \in E_G$.

It is important to note that the vertex $\alpha(e) \in V_G$ is called the *origin* or *source vertex* of the edge $e \in E_G$; similarly the vertex $\beta(e) \in V_G$ is called the *end* or *tail vertex* of the edge $e \in E_G$, [15].

A graph may be *directed* or *undirected*. A directed graph, also known as a digraph is called a *triple*, if it is of this form $G = (V_G, E_G, \Theta)$. In such a case it is safe to say there is a source vertex and an end vertex in the directed graph G; where Θ is the *incidence mapping* from the edge set E_G to the Cartesian product of the vertex sets $V_G \times V_G$, [2].

Let us refer to Figure 1.1 for a clear understanding. It is clear that in $e_1 = \Theta(e_1) = (u, v) = (\alpha(e_1), \beta(e_1))$, and vertex $u = \alpha(e_1)$ is the source vertex of the edge e_1 and the vertex $v = \beta(e_1)$ is the end vertex of the edge e_1. Similarly, in $e_2 = \Theta(e_2) = (v, w) = (\alpha(e_2), \beta(e_2))$ the vertex $v = \alpha(e_2)$ is the source vertex of the edge e_2, and $w = \beta(e_2)$ is the end vertex of the edge e_2.

Here, one can say that the edges (u, v) and (v, u) are distinct edges in the edges set E_G. However, an undirected graph may have that the edges (u, v) and (v, u) are the same edges in the edges set E_G, i.e. $(u, v) = (v, u)$, [2].

The *cardinality* of the vertex set of a graph is the number of vertices in a graph, also known as the order of the graph. And the number of edges that connects vertices in the graph is called the size of the graph. These are denoted by $|V_G| = n_G, \ \forall n_G \in \mathbb{N}_{n_G \neq 0}$ as the order of the graph and $|E_G| = m_G, \ \forall m_G \in \mathbb{N}$ as the size of the graph [2].

A *trivial graph* is a graph of order equal to one, i.e. $|V_G| = 1$ and a *non-trivial graph* is a graph of order greater or equal to 2, i.e. $|V_G| \geq 2$

A *graph* is called an empty graph if the size of the graph is zero, that is the graph has no edges or $|E_G| = 0$. However, a graph is non-empty if $|E_G| \geq 1$.

A graph with order $|V_G| = n_G$ and size $|E_G| = \binom{n_G}{2} = \frac{n_G(n_G-1)}{2}$ is called a complete graph K_{n_G}. Here, each pair of distinct vertices has an edge [12].

Refer to the following example:

Let $|V_G| = 5$ and $|E_G| = \binom{5}{2} = \frac{5(5-1)}{2} = \frac{5\times 4}{2} = \frac{20}{2} = 10$. Then $G = K_5$.

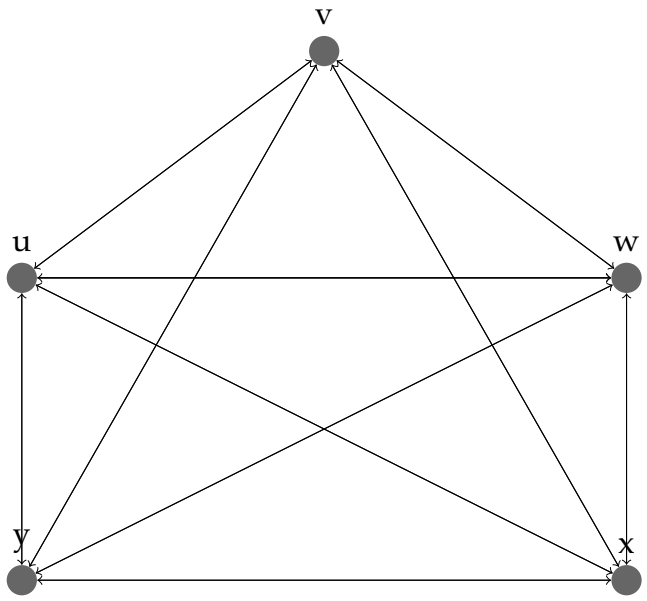


Figure 1.2: K_5 Graph

For any other type of graph that has order $|V_G| = n_G$ and has size $|E_G| = m_G$, the number of edges (links) in that graph lies within the bounds $0 \leq m_G \leq \binom{n_G}{2} = \frac{n_G(n_G-1)}{2}$, [12].

Vertices are *adjacent* to each other if they share an edge; refer to Figure 1.

$$\Theta(e_1) = (\alpha(e_1), \beta(e_1)) = (u, v)$$
$$\Theta(e_2) = (\alpha(e_2), \beta(e_2)) = (v, w)$$

It is clear that the vertices u and v are adjacent to each other since they share an edge; such vertices are referred to as neighbors. Likewise, vertices v and w are adjacent to each other [2].

A set of vertices is called a *set of neighbors* of a distinct vertex u, if all vertices in the set are adjacent to the vertex u. The set of neighbors is called an *open neighborhood* of the vertex u and is denoted by $N_G(u)$.

The *closed neighborhood* of the vertex u is denoted by $N[u] = N_G(u) \cup u$; this neighborhood includes all the vertices adjacent to vertex u and itself [2].

A vertex may be connected to itself via an edge called a *loop*. Here, the source vertex of this edge is the same as the end vertex of that edge; see Figure 1.3 [2].

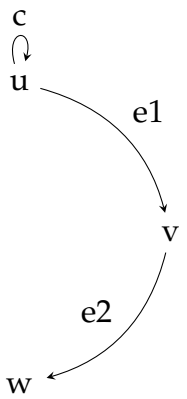


Figure 1.3: Directed Graph with 3 vertices

Counting the *number of edges* in an edge set E_G of a graph G, such that the edges in E_G are incident with a vertex u, is defined as the *degree* of a vertex u in the graph G, and is denoted by $deg_G(u)$ [2].

The *degree* of a vertex u may also be defined as the number of vertices in an open neighborhood of the vertex u, i.e. $deg_G(u) = |N_G(u)|$, [12].

There are different types of vertices and edges with distinct properties and roles a graph could contain. Graphs may contain vertices that are incident to no edges or incident to one edge, these vertices are called *isolated vertices* and *end-vertices*, respectively. An edge is known to be a *pendant-edge* if the edge is incident with an end-vertex.

Digraphs contain directed edges, and their respective vertices have vertex degrees where the directed edges are directed towards the vertices and directed outward the vertices; these are called *in-degree* of a vertex and *out-degree* of a vertex, respectively. In the following detailed definition, refer to Figure 1.1.

Let G be a directed graph, then the edge set of the graph contains directed edges; and the *in-degree* of a vertex in the graph is the number of directed edges pointing towards the vertex and the *out-degree* of a vertex in the graph is the number of directed edges pointing outwards from the respective vertex in the graph. It follows from [13] that,

$$indeg(w) = |\{e \in E_G : \beta(e) = w\}|$$

$$outdeg(u) = |\{e \in E_G : \alpha(e) = u\}|$$

The *maximum degree* of a graph G is known as the highest degree among the vertices within the graph G. Similarly, the *least degree* of a graph G is the smallest degree between the vertices within the graph. The lowest degree is denoted by $\sigma(G)$ and the highest degree is denoted by $\triangle(G)$.

It is clear that $0 \leq \sigma(G) \leq deg_G(u) \leq \triangle(G)$ is true for a graph G [2].

If a graph G has n_G number of vertices and m_G number of edges, then the total sum of degrees of all vertices in a graph G is twice the number of edges, see [2],

$$\sum_{i=1}^{n_G}(deg_G(u_i)) = 2m_G.$$

We shall use the following as a corollary:

The number of vertices in a graph G with *odd degree* is even [2].

A subgraph $G_1 = (V_{G_1}, E_{G_1})$ of a graph $G = (V_G, E_G)$ is a graph with vertex set V_{G_1} contained in the vertex set V_G of graph G,

$$i.e.\ V_{G_1} \subset V_G;$$

and with an edge set E_{G_1} contained in the edge set E_G of the graph G,

$$i.e.\ E_{G_1} \subset E_G.$$

It is not necessary for a subgraph to contain all possible edges of a graph [13].

1.2 Graph Matching

In this section we define and examine the fundamental concepts for graph matching. The definitions introduced in this section were developed by Gabriele Valiente in [21] and were in turn used in various other papers for detecting whether two given graphs are matches of each other; and or they contain subgraphs that match. Graph matching is a procedure used to detect similarities between vertex sets and edge sets of two given graphs by satisfying some given similarity conditions such that vertex adjacency and non-adjacency in the two graphs are preserved, [5].

In this dissertation we shall consider a detailed description of two important matching categories; that is the exact and inexact matching procedures. An exact graph matching procedure employs a mapping or relation function which maps vertices from a vertex set V_Q onto another vertex set V_G of two distinct graphs Q and G, respectively; such that if two adjacent vertices $v_1, v_2 \in V_Q$ are mapped onto two adjacent vertices $u_1, u_2 \in V_G$ in the other graph, adjacency is preserved.

Similarly, if we have two non-adjacent vertices $v_3, v_4 \in V_Q$ contained in the graph Q, then these two vertices should be mapped onto two non-adjacent vertices $u_3, u_4 \in V_G$ in the other graph G, i.e., non-adjacency must be preserved. There are several distinct types of exact matching procedures which were previously studied, some of the exact matching procedures are listed below:

- Graph Isomorphism - The most strict exact matching procedure.
- Subgraph Isomorphism - The least strict exact matching procedure.
- Monomorphism - The slightly weaker exact matching procedure.
- Homomorphism - The weaker exact matching procedure.

We will go through a detailed description of the above mentioned exact matching procedure, however, our main focus will be the subgraph isomorphism procedure.

As described in [21], two given graphs are said to be exact matches under isomorphism if there exists a one-to-one relation between their given sets of vertices that maintains adjacency and non-adjacency between the vertex sets. The concepts introduced in this sections will be used in the sections to follow.

Let $Q = (V_Q, E_G)$ and $G = (V_G, E_G)$ be two graphs with n_Q and n_G vertices, respectively. These two given graphs are said to be *isomorphic* provided that there exists a one-to-one relation/ mapping $R \subseteq V_Q \times V_G$ such that, for each vertex couple $v_k, v_j \in V_Q$ and $u_k, u_j \in V_G$ with relations $(v_k, u_k) \in R$ and $(v_j, u_j) \in R$; the pair (v_k, v_j) is said to be adjacent in Q if and only if the pair (u_k, u_j) is also adjacent in G. Note that, in such cases the relation R is called the relation set of all graph isomorphisms from Q to G.

Below is an example of two isomorphic graphs, note that the objective of the isomorphism problem is to test if two given graphs are the same. However, two isomorphic graphs can be structured to appear as though they are different; that is to say, the graphs vertices and edges

can be labeled differently.

Example

Just as stated in [21], the two given graphs Q and G are said to be isomorphic; the mapping $R \subseteq V_Q \times V_G = \{(v,a),(z,b),(w,c),(x,d),(y,f),(u,g)\}$ shows the relation of vertices between the two given graphs. See Figure 1.4 and Figure 1.5 which depict two isomorphic graphs whose vertices are labeled differently and the shapes are differently.

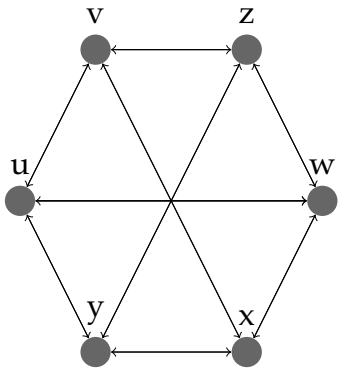


Figure 1.4: $Q = (V_Q, E_Q)$ Pattern Graph

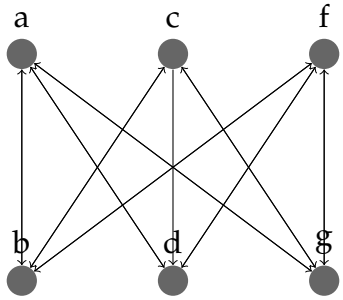


Figure 1.5: $G = (V_G, E_G)$ Data Graph

Gabriel Valiente constructed an essential algorithm for detecting isomorphism between two given graphs. In [21], an efficient and sufficient backtracking procedure is employed to develop a straightforward algorithm for detecting isomorphism between two given graphs. The algorithm satisfies the conditions that two graphs are said to be isomorphic if and only if there exists a one-to-one mapping among their respective vertex sets such that adjacency and non-adjacency is preserved. Below is a brief description of graph isomorphism constructed in [21].

Let the two given graphs be denoted by $Q = (V_Q, E_Q)$ and $G = (V_G, E_G)$, and consider two subsets $V_{Q_s} \subseteq V_Q$ and $V_{G_s} \subseteq V_G$. Let $R \subseteq V_{Q_s} \times V_{G_s}$ be an *isomorphism relation* between the subgraph Q_s of Q induced by the vertex set V_{Q_s} onto the subgraph G_s of G induced by the vertex set V_{G_s}.

For each vertex v contained in the vertex set V_Q without V_{Q_s}; and for each vertex u contained in the vertex set V_G without V_{G_s}, it should be noted that the mapping $R \cup \{(v, u)\}$ is an isomorphism of the subgraph Q_s of the graph Q; which is made up of a vertex set denoted by $V_Q \cup \{v\}$, onto the subgraph G_s of G; which is also made up of the vertex set $V_G \cup \{u\}$, if and only if the below mentioned constraints are satisfied for each and every vertex x contained in V_{Q_s} and y contained in V_{G_s}; and note that $(x, y) \in R$, [21] :

- The graph Q contains an edge (x, v) in the edge set E_Q if and only if the graph G also contains an (y, u) in the edge set E_G.
- The graph Q contains an edge (v, x) in the edge set E_Q if and only if the graph G also contains an (u, y) in the edge set E_G.

Below is a detailed proof developed in [21] along with a diagram used to visualize the isomorphic mapping $R \cup \{(v, u)\}$, as discussed above.

Proof. Consider two graphs denoted by $Q = (V_Q, E_Q)$ and $G = (V_G, E_G)$ with their respective vertex sets V_Q, V_G and edge sets E_Q, E_G. We denote by $V_{Q_s} \subseteq V_Q$ and $V_{G_s} \subseteq V_G$ the vertex sets of the subgraphs Q_s and G_s. Let the mapping $R \subseteq V_{Q_s} \times V_{G_s}$ identify an isomorphism which maps vertices from the vertex set of the subgraph Q_s onto the vertex set of the subgraph G_s.

We assume that $(x, v) \in E_Q$ if and only if $(y, u) \in E_G$ for each and every relation $(x, y) \in R$. Similarly, assume that $(v, x) \in E_Q$ if and only if $(u, y) \in E_G$ for each and every relation $(x, y) \in R$.

It is clear that the relation $R \cup (v, u)$ is an isomorphism of the subgraph Q_s of graph Q generated by $V_{Q_s} \cup \{v\}$ onto the subgraph G_s of graph G generated by $V_{G_s} \cup \{u\}$.

To prove the converse, we let $R \cup (v, u)$ be a subgraph isomorphism between the subgraph Q_s of graph Q and the subgraph G_s of graph G, induced by $V_{Q_s} \cup \{v\}$ and $V_{G_s} \cup \{u\}$, respectively.

Let $x \in V_{Q_s}$ and $y \in V_{G_s}$ such that $(x, y) \in R$. Because there are relations $(z, v), (x, y)$ in $R \cup \{(v, u)\}$, it is satisfied that $(x, v) \in E_Q$ if and only if $(y, u) \in E_G$; similarly, $(v, x) \in E_Q$ if and only if $(u, y) \in E_G$. ■

Let $G = (V_G, E_G)$ and $Q = (V_Q, E_Q)$ be two graphs; the graph Q is an isomorphism of graph G if and only if :

There exists a bijective map $\delta : V_Q \mapsto V_G$ such that there is an edge $(v, u) \in E_Q$ if and only if $(\delta(v), \delta(u)) \in E_G$.

We can also verify isomorphism using a matrix P called the permutation matrix, which is made up of all permutations from vertex set V_Q to vertex set V_G. The bijective map δ is called a permutation if the map specifies a one-to-one correspondence between vertices.

The two graphs $G = (V_G, E_G)$ and $Q = (V_Q, E_Q)$ are said to be isomorphic if and only if $B_G = PA_QP^T$ for some permutation matrix P, where B_G and A_Q are the adjacency matrices of graph G and graph Q, respectively.

Consider the two given graphs in Figure 1.6 and Figure 1.7,

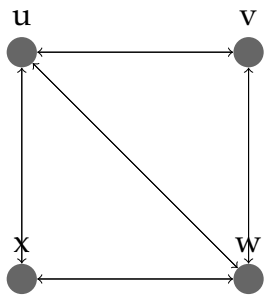


Figure 1.6: $Q = (V_Q, E_Q)$ Pattern Graph

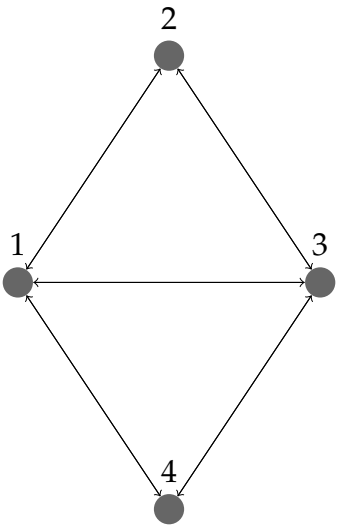


Figure 1.7: $Q = (V_Q, E_Q)$ Pattern Graph

We can test for isomorphism between the two given graphs by using a bijective map denoted by δ.

$$\delta : V_Q \mapsto V_G$$

Below we represent the bijective map δ as a permutation from V_Q to V_G by mapping vertex u to vertex 3, vertex v to vertex 4, vertex w to vertex 1, and vertex x to vertex 2.

$$\delta = \begin{pmatrix} u & v & w & x \\ 3 & 4 & 1 & 2 \end{pmatrix}.$$

The objective here, is to verify whether δ specifies a one-to-one correspondence (one-to-one relation). It is clear that if $(u, v) \in E_Q$, then the edge $(\delta(u), \delta(v)) = (3, 4)$ must also be an edge in E_G for a one-to-one relation to be specified. Similarly, if $(w, x) \in E_Q$ then $(\delta(w), \delta(x)) = (1, 2)$ must also be an edge in E_G for a one-to-one relation to be specified.

Furthermore, if $(v, x) \notin E_Q$ then $(\delta(v), \delta(x)) = (4, 2) \notin E_G$. In a general case, we say that adjacency and non-adjacency between vertices in both graphs needs to be satisfied.

Now, we write each permutation δ in matrix form to construct the permutation matrix P. That is to say,

$$P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Where each entry in P represents a relation, zero means there is no relation between vertices and one means that there is a relation between vertices. Note that the permutation matrix is constructed with respect to some given permutation δ from vertex set V_Q to vertex set V_G.

Below we define what a graph homomorphism is and how it occurs.

Let G and G_1 be graphs with vertex sets V_G, V_{G_1} and edge sets E_G, E_{G_1} respectively. Then a *graph homomorphism* is defined to be a mapping γ from the vertex set V_G to a vertex set V_{G_1} if and only if:

(1) For each vertex u in the vertex set V_G, mapping this vertex yields a vertex $\gamma(u)$ in the vertex set V_{G_1}.

(2) For each vertex u_1 in the vertex set V_{G_1}, there must exist a vertex $u^{'}$ inside the vertex set V_G such that mapping $u^{'}$ yields the vertex u_1, i.e., $\gamma(u^{'}) = u_1$.

(3) For each edge (u, v) in the edge set E_G and images $\gamma(u) \neq \gamma(v)$, there must exist an edge $(\gamma(u), \gamma(v)) \in E_{G_1}$.

(4) For each edge (u_1, v_1) in the edge set E_{G_1}, there exists vertices $u^{''}$, $v^{''} \in V_G$ such that an edge $(u^{''}, v^{''})$ is contained in the edge set E_G and $\gamma(u^{''}) = u_1$ and $\gamma(v^{''}) = v_1 \iff$ [15].

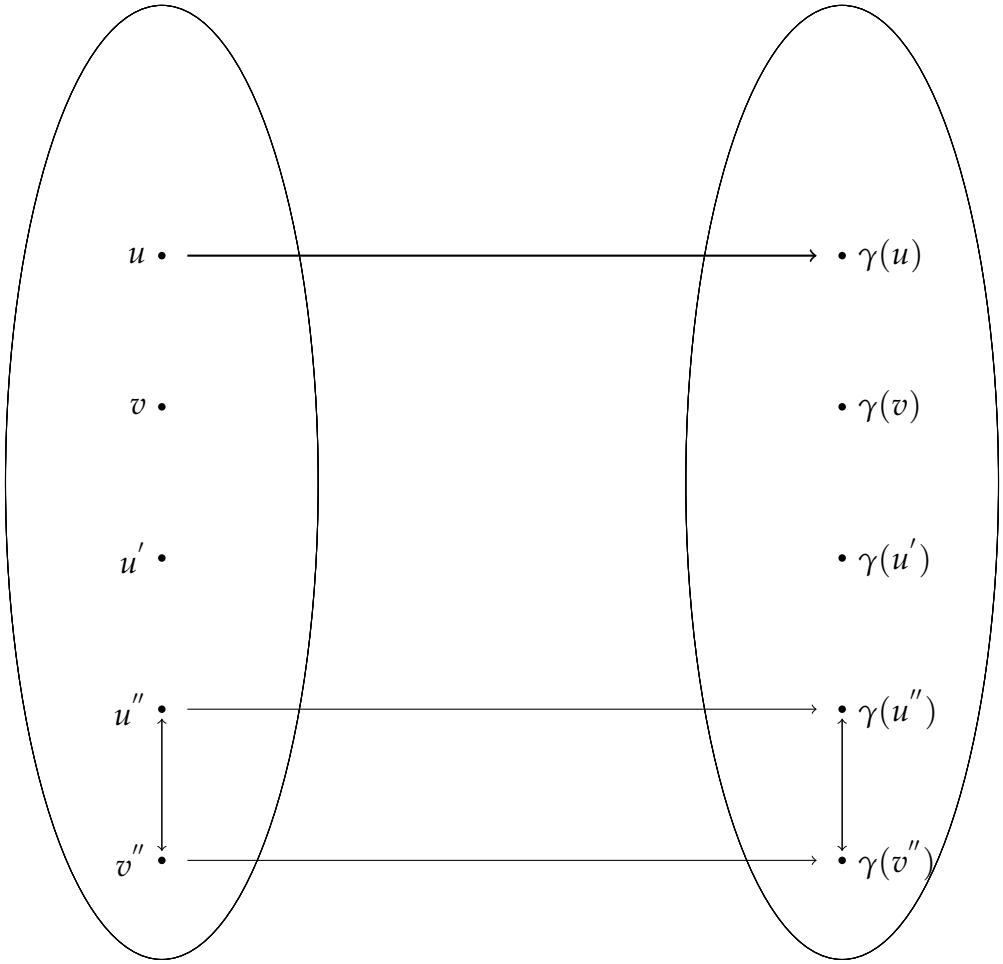


Figure 1.8: Graph Homomorphism

In the mathematics of graph theory, graph homomorphism is a mapping that preserves the edges between two given graphs; however, the mapping need not be one to one. Refer to the above definition, γ is a mapping that preserves adjacency between edges of the graphs. Refer to Figure 1.8, for a clear visualization of this definition.

Chapter 2

2.1 The Ullmann Subgraph Isomorphism Problem

As mentioned in the previous section, graphs are used for modelling a wide range of application problems such as representing biological networks at the molecular, protein, or species level. The most significant generalization of the Graph Isomorphism Problem is known as the Subgraph Isomorphism Problem [21]. In previous research [18], the Subgraph Isomorphism Problem was applied to biochemical data to detect matches between a given chemical compound (i.e., graph) and the sub-chemical compounds (i.e., subgraphs) of a larger chemical compound (i.e., larger graph). The important question is how does one detect all matches of a given pattern graph in a given data graph. Achieving these graph matches is instinctively hard since it is an NP-complete decision problem.

The main purpose of this section is to understand and develop rational and coherent techniques for finding subgraphs in a large data graph that might be isomorphic matches to a given pattern graph. Several other strategies developed by previous researchers considered algorithms built on filtering and verification; which help in reducing the computational processing costs [9]. In this section we explore in detail the techniques developed by Ullmann in [19] for subgraph isomorphism detection; and in the sections to follow we shall compare Ullmanns algorithm to other techniques developed for subgraph isomorphism detection.

In this section of the dissertation we define the Subgraph Isomorphism Problem just as J. R. Ullmann defines it in his paper [19]. The Subgraph Isomorphism Problem states that "A graph $Q = (V_Q, E_Q)$ is said to be isomorphic to a subgraph $G_s = (V_s, E_s)$ of a given graph $G = (V_G, E_G)$ if and only if there exists a one-to-one correspondence between the vertex set of the graph Q and the vertex set of the subgraph G_s contained in the large graph G, which can maintain adjacency and non-adjacency", [19].

As noted by Ullmann in [19], this one-to-one correspondence is denoted by a permutation matrix P^d which is used as a binary matrix that specifies a relation between vertices in the pattern graph and vertices in the data graph.

By definition, a (standard) *subgraph isomorphism* between a graph Q and a graph G is called a one-to-one map $\delta : V_Q \mapsto V_G$ that satisfies the condition below [3]:

$$(v_r, v_c) \in E_Q \implies (\delta(v_r), \delta(v_c)) \in E_G \tag{2.0}$$

Likewise, an *induced subgraph isomorphism* between a graph Q and another graph G is called a one-to-one map δ from the vertex set of the graph Q onto the vertex set of the graph G satisfying the condition in Equation (2.0), see [4].

The above definition, in Equation (2.0) states that for each vertex pair (v_r, v_c) contained in the vertex set V_Q there exists a vertex pair $(\delta(v_r), \delta(v_c))$ contained in the vertex set V_G such that $\{v_r, \delta(v_r)\}$ and $\{v_c, \delta(v_c)\}$ show a one-to-one correspondence in the permutation matrix P^d. That is to say that the vertex pair $(\delta(v_r), \delta(v_c))$ is contained in E_G if the vertex pair (v_r, v_c) is contained in E_Q.

J. R. Ullmann constructed an algorithm for the Subgraph Isomorphism Problem in [19], which is used as the framework in other existing isomorphism detection techniques. In this section we will describe in detail the design of Ullmann's simple enumeration algorithm for subgraph isomorphism detection which was developed in [19], and we shall also describe certain improvements that could be enforced [3].

In the simple enumeration algorithm, we initialized a permutation matrix P^0 whose element entries are ones and zeros and shall be used as an input for the enumeration algorithm. The permutation matrix P^0 is used simultaneously as a representation for all possible mappings, not individual mappings of the pattern graph to the data graph. That is, the initialized permutation matrix P^0 is seen as the reference matrix for generating all other possible candidate permutation matrices for isomorphism detection. This initial permutation matrix is constructed by comparing the degrees of vertices $deg_Q(v_r)$ in the pattern graph Q to the degrees of vertices $deg_G(v_c)$ in the data graph G.

Since the initialized permutation matrix P^0 is of great significance for the simple enumeration algorithm, we shall take a closer look at how it is constructed. Let us first consider the note below,

> **Note1:** A vertex $v_c \in V_G$ in the data graph is a possible candidate for a vertex $v_r \in V_Q$ in the pattern graph if the degree of vertex $v_r \in V_Q$ is less than or equal to the degree of vertex $v_c \in V_G$, [4].

which implies that many vertices contained in the data graph have more adjacent vertices as compared to the pattern graph. The initialized matrix is constructed by satisfying the condition below, which is called the degree criterion [19]:

$$p^0_{r,c} = \begin{cases} 1, & if\ deg_G(v_c) \geqslant deg_Q(v_r) \\ 0, & otherwise \end{cases} \tag{2.1}$$

In the Ullmann simple enumeration algorithm, the initialized matrix P^0 is known as the candidate permutation matrix for subgraph isomorphism detection at depth zero. See below,

$$V_Q \left\{ \underbrace{\begin{bmatrix} p^0_{1,1} & p^0_{1,2} & p^0_{1,3} & \cdots & p^0_{1,n_G} \\ p^0_{2,1} & p^0_{2,2} & p^0_{2,3} & \cdots & p^0_{2,n_G} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p^0_{n_Q,1} & p^0_{n_Q,2} & p^0_{n_Q,3} & \cdots & p^0_{n_Q,n_G} \end{bmatrix}}_{V_G} \right. .$$

It is important to note that the row entries of the permutation matrix P^0 correspond to all of the vertices contained in the vertex set V_Q, and its column entries corresponds to all the vertices contained in the vertex set V_G. As shown in Equation (2.1), we can say that if the entry $p^0_{r,c} = 1$, then there exists a correspondence between the vertex v_r contained in V_Q and the vertex v_c contained in V_G. However, if the entry $p^0_{r,c} = 0$ then there is no correspondence between the vertex $v_r \in V_Q$ and the vertex $v_c \in V_G$.

Note2: The number of vertices in the pattern graph may be less than or equal to those in the data graph. That is, $n_Q \leq n_G$, see [19].

As mentioned above, the initialized permutation matrix P^0 is used to construct all other possible candidate permutation matrices P^d where d is any depth of vertices in the corresponding candidate permutation matrix for subgraph isomorphism detection.

All these candidate permutation matrices P^d at depth d are constructed from the initial permutation matrix P^0 by systematically changing all the 1 entries to zero, except one 1 of every row and every column of the initial matrix $P^0 = [p^0_{r,c}]$; which in turn satisfies the condition which states that "all possible candidate permutation matrices P^d at depth d should contain exactly one 1 in each of its row entries and at most one 1 in each of its column entries" [19]. And this is done in a step-by-step fashion using the backtracking procedure.

- The entries must be $0's$ and $1's$,

  i.e., $p^d_{r,c} \in \{0, 1\}, \ \forall r \in \{1, \ldots, n_Q\}, c \in \{1, \ldots, n_G\}$;

- The sum of all entries in a column must also equate to 1 for all columns [19],

$$\sum_{r=1}^{n_Q} p^d_{rc} = 1 \ \ \forall c \in \{1, \ldots, n_G\}; \tag{2.2}$$

- The sum of all entries in a row must equal 1 for all rows, as shown in [19],

$$\sum_{c=1}^{n_G} p^d_{rc} = 1 \ \ \forall r \in \{1, \ldots, n_Q\}. \tag{2.3}$$

There are $n_G(n_G-1)(n_G-2)\ldots(n_G-n_Q+1)$ possible candidate permutation matrices (for subgraph isomorphism detection) that could be constructed and be tested if they indeed specify an isomorphism between two given graphs; say the pattern graph Q and the data graph G, [8]. It is important to note that the Ullmann's algorithm detects subgraph isomorphism by using a trial and error method called the Brute-Force enumeration procedure. As observed in [19], Ullmann noted that for extremely large graphs, the Brute-Force enumeration procedure is inefficient since it rapidly fails to detect isomorphic matches [4].

We shall study the Ullmann's algorithm and try to further improve the algorithms efficiency by focusing on the time complexity of the algorithm and also the number of isomorphic matches it detects; however, we also look at the number of partial matches it detects.

Among all of these possible candidate permutation matrices P^d, each matrix gives rise to its respective candidate mapping from the vertex set V_Q of the pattern graph to the vertex set V_G of the data graph. In [19], J. R. Ullmann constructs a set of candidate vertices such that for each iterative step the algorithm picks a pair (v_r, v_c) of vertices that have not been compared, one vertex from the vertex set V_Q at the r_{th} row, (this is a vertex associated with the pattern graph Q) and the other vertex from the vertex set V_G satisfying the condition that $p^d_{r,c} = 1$ for each vertex selected from V_G, see [4].

It is significant to note that the algorithm does not select the vertices in the data graph G one by one, instead it tries all the vertices that have a 1 in the initial permutation matrix P^0 [3],

$$C^d(v_r) = \{v_c \in V_G | p^d_{r,c} = 1\} \tag{2.4}$$

The set of candidate vertices in Equation (2.4) is constructed by selecting all the vertices contained in the vertex set V_G such that there exists a one-to-one correspondence between each vertex contained in the vertex set V_Q and each vertex contained in the vertex set V_G, as discussed in [4].

Since the Ullmann simple enumeration algorithm is executed iteratively and also uses the backtracking procedure, we can recursively advance from depth d to the next depth $d+1$; and reversing the procedure may be represented by $d-1$, [8]. Note that the algorithm advances from a candidate permutation matrix P^d at depth d to another candidate permutation matrix P^{d+1} at depth $d+1$.

Recall that in the process of constructing the possible candidate permutation matrices, we need to start with the first vertex (i.e., an arbitrary vertex in graphs and/or a root-vertex in a tree) of the pattern graph and try all possible assignments (row-entry assigned to column-entry), which have a 1 in the initial permutation matrix P^0, [3].

In this step of the algorithm we assign a vertex in row r (from the pattern graph) to a vertex in column c (from the data graph). To satisfy the one-to-one correspondence condition, we assign a vertex $v_r \in V_Q$ to exactly one candidate vertex $v_c \in V_G$. Hence, there are no other compatible vertices for the vertex in r except at column c. So we can set the entire row r to 0, except of course at the position c, [4]. The same applies to the column c since it is already assigned to

r; so we can set all other entries in column c to 0, [4].

We construct the initial candidate permutation matrix P^0 by first constructing the adjacency matrix of the pattern graph Q and the adjacency matrix of the data graph G. This will help with easily comparing the degrees of vertices contained in the pattern graph to the degrees of vertices contained in the data graph [3]. Therefore, by satisfying the condition in equation (2.1) we are able to construct the initial candidate permutation matrix P^0.

In [19], the Ullmann's algorithm rearranges the rows and columns of the adjacency matrix of the data graph G by simultaneously multiplying the adjacency matrix of graph G on left and on the right by each possible candidate permutation matrix contained in the set χ of all possible candidate permutation matrices. All these candidate permutation matrices contained in χ are constructed for subgraph isomorphism detection. The set of all possible candidate permutation matrices χ is of the form $\{P^0, P^1, P^2, \ . \ . \ .\}$, [3].

As mentioned earlier in this section, we shall first construct the adjacency matrices for both the pattern graph and the data graph, which will be used in subgraph isomorphism detection. The adjacency matrix of the pattern graph is denoted by $A_Q = [a_{rc}]$ and defines an $n_Q \times n_Q$ square matrix; similarly, the adjacency matrix of the data graph is denoted by $B_G = [b_{rc}]$ and is an $n_G \times n_G$ square matrix with the condition that $n_Q \leq n_G$, [17].

However, note that in Listing 2.2, Listing 2.3 and Listing 2.4 we use A and B to represent the adjacency matrices for pattern and data graph, respectively. As stated by Ullmann in [19], we need to construct all possible candidate permutation matrices and then verify if they indeed specify an Subgraph Isomorphism from the pattern graph to the data graph [8]. This is done by applying the Brute-Force enumeration procedure.

In Listing 2.2 and 2.3 we use an interpreted high-level programming language called Python for the implementation to show how we construct a pattern graph Q and data graph G along with their adjacency matrices A_Q and B_G, respectively. And in Listing 2.4 we show how the initial candidate permutation matrix P^0 is constructed from the adjacency matrices A_Q and B_G.

In this section we will further explain how these possible candidate permutation matrices are constructed by the use of the original Ullmann simple enumeration algorithm which was formulated in [19]. And we shall further give a flow diagram of the original Ullmann simple enumeration algorithm [18], which we then simplify for a clear understanding of the Ullmann's algorithm. Note that in the flow diagram we will use the notation derived in [18], which was also used in the original paper by Ullmann in [19].

We shall also look into the details of the algorithm employing the refinement procedure, that is, the algorithm which was constructed in [19] to reduce the amount of computation required for finding the subgraph isomorphisms [3]. We shall also enforce the improvements formulated by Uroš and Jurij in [4].

We consider a very simple example to illustrate how these possible candidate permutation matrices P^d are constructed. In this example we consider a labelled pattern graph of order five and size five as given below, and its respective adjacency matrix A_Q.

Listing 2.1: Construction of the pattern Graph Q

```
import networkx as nx
import numpy as np
import random
import matplotlib.pyplot as plt

Q = nx.Graph()
EdgeSet = [[0, 1],[0, 2],[0, 3],[1, 3],[2, 4]]
Q.add_edges_from(EdgeSet)
n_nodes = 5

labels = {}
labels[0] = '$a$'
labels[1] = '$b$'
labels[2] = '$c$'
labels[3] = '$d$'
labels[4] = '$e$'

pos = {i :( random.randint(0,50),random.randint(0,100)) for i in range(n_nodes)}
nx.draw(G, pos, nodelist=[0, 1, 2, 3, 4], edge_labels=True)
nx.draw_networkx_labels(G, pos, labels, font_size=16)
Q.degree([0, 1, 2, 3, 4])
A = nx.attr_sparse_matrix(Q, rc_order =[0,1,2,3,4])

A.todense() #Adjacency Matrix for the pattern graph Q.
print(A.todense())
```

We generate a simple pattern and data graph along with their respective adjacency matrices using the codes written in Listing 2.2, 2.3 and 2.4. We use these adjacency matrices to construct the initialized candidate matrix by satisfying the condition in Equation (2.1).

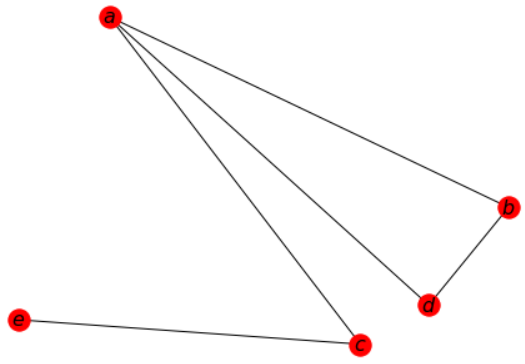


Figure 2.1: Pattern Graph Q

The pattern graph in Figure 2.1 has an adjacency matrix defined in the equation below,

$$A_Q = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.5}$$

Listing 2.2: Construction of the large Data Graph G

```
import networkx as nx
import numpy as np
import random
import matplotlib.pyplot as  plt

G = nx.Graph()
EdgeSet = [[0,  1],[0,  2],[0,  10],[1,  10],[1,  9],[1,  3],[2,  5],[3,  6],[5,  4],[5,  8],[5,7],
          [7,  6],[6,  9],[6,  8],[6,  10]]
G.add_edges_from(EdgeSet)
n_nodes = 11

labels  = {}
labels[0]  = '$a$'
labels[1]  = '$b$'
labels[2]  = '$c$'
labels[3]  = '$d$'
labels[4]  = '$e$'
labels[5]  = '$f$'
labels[6]  = '$g$'
labels[7]  = '$h$'
labels[8]  = '$i$'
labels[9]  = '$j$'
labels[10]  = '$k$'

pos = {i :( random.randint(0,50),random.randint(0,100)) for i  in  range(n_nodes)}
nx.draw(G, pos, edge_labels=True)
nx.draw_networkx_labels(G, pos ,labels,  font_size=16)

G.degree([0,  1,  2,  3,  4])
B = nx.attr_sparse_matrix(G, rc_order   =[0,1,2,3,4,5,6,7,8,9,10])

B.todense() #Adjacency Matrix for the data graph G.
print(B.todense())
```

We further consider a larger graph G called the data graph in Figure 2.2 with order 11 along with its adjacency matrix denoted by B_G.

$$B_G = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.6}$$

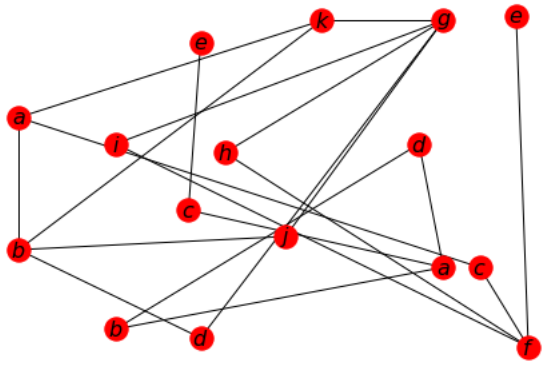


Figure 2.2: Data Graph G

The condition defined in Equation (2.1) was constructed by Ullmann in [19], and it assits us a lot in constructing the initial candidate matrix P^0 for subgraph isomorphism detection using the two given graphs constructed above, in Figure 2.1 and Figure 2.2.

Below is the code we used to construct the initial candidate matrix P^0:

Listing 2.3: Construction of the initialized candidate matrix P^0

```
def Initial_Candidate_Matrix(A, B):
    a = np.asarray(A)
    b = np.asarray(B)
    P = np.zeros((a.shape[0],b.shape[1]))
    for i in range(a.shape[0]):
        for j in range(b.shape[0]):
        if a[a[i,:] !=0].shape[0] <= b[b[j,:] !=0].shape[0]: #Condition in equation 1
                P[i, j] = 1
            return (P)
```

$$P^0 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{2.7}$$

We shall now go through an in-depth analysis of the original Ullmann simple enumeration algorithm. Since this algorithm is based on the depth first tree search algorithm, we shall give a complete analysis of the original Ullmann algorithm and how it works.

Explaining the original Ullmann simple enumeration algorithm...

In the original Ullmann simple enumeration algorithm, we first start by setting up an $n_Q \times n_G$ initial candidate permutation matrix for subgraph isomorphism detection by using the degree

criterion in Equation (2.1). In this dissertation we denote that initial candidate permutation matrix by P^0.

After constructing the initialized candidate permutation matrix P^0, we use it to generate all other possible candidate permutation matrices that could be used for the subgraph isomorphism detection. This algorithm is used to both traverse and search through the initial candidate permutation matrix in order to satisfy isomorphism conditions.

To begin the implementation of the algorithm one needs to first pick a reference vertex contained in the pattern graph. This is done by selecting which row in P^0 to start with [4]. We shall start at the first row of the candidate permutation matrix P^0.

That is, we first select an arbitrary vertex in a graph (the pattern graph) and/or start at a root-vertex in a tree (the pattern graph), and then we recursively move through all the other vertices contained in the pattern graph in order to assign vertices that are contained in the data graph. This assignment must satisfy a one-to-one correspondence between the vertices in the pattern graph and those in the data graph [4].

The one-to-one correspondence condition assists with constructing all other possible candidate permutation matrices that will be used to detect the subgraph isomorphism between the two given graphs. It is significant to note that in mathematics, a one-to-one correspondence refers to a condition whereby vertices contained in the vertex set V_Q can be evenly matched with the vertices contained in the vertex set V_G, [2].

"Evenly matched" implies that each vertex contained in the vertex set V_Q must be paired with one and only one vertex contained in the vertex set V_G. And the converse must also be true, i.e., each vertex contained in the vertex set V_G must be paired with one and only one vertex contained in the vertex set V_Q, and none of the vertices from either vertex set must be left unpaired [15].

As a result for one-to-one correspondence, we can conclude that each vertex contained in the vertex set V_Q should be paired with exactly one vertex contained in the vertex set V_G, and each vertex contained in the vertex set V_G should be paired with exactly one vertex contained in the vertex set V_Q. A one-to-one correspondence is signified by a bijective map between the two vertex sets [2], i.e., $\delta : V_Q \mapsto V_G$ as defined in Equation (2.0).

Recall that the rows of the initial candidate permutation matrix P^0 correspond to the vertices contained in the pattern graph and the columns of P^0 correspond to the vertices in the data graph [19]. In the original Ullmann simple enumeration algorithm a vertex is selected at depth d = 1; this is a vertex in the initial candidate matrix P^0 at row one. The depth of a vertex may be thought as selecting a vertex in the row of the initial candidate matrix P^0, in order to assign the selected vertex in the row of P^0 to the vertex in the column of P^0. The simple enumeration algorithm measures depth from d = 1 up to the order of the pattern graph, that is $d = n_Q$ [19].

If the depth is less than the order of the pattern graph, $d < n_Q$, this implies that we are dealing with non-terminal vertices in the pattern graph [19]. Each non-terminal vertex corresponds to candidate permutation matrices that are distinct from the initialized candidate permutation

matrix in such a way that in depth d, all of the one entries of d rows are changed to zero except one 1 [19].

All the terminal vertices are at depth $d = n_Q$ in the pattern graph, and correspond to distinct candidate permutation matrices that still need to be verified whether an isomorphism is specified.

For clarity purposes, Ullmann constructed the algorithm such that the depth $d = 1, 2, \ldots, n_Q$ is equivalent with the $1^{st}, 2^{nd}, \ldots, n_Q^{th}$ rows of the adjacency matrix A_Q of the pattern graph Q, respectively. However, in the experiment Ullmann conducted for subgraph isomorphism, the depth was employed in such a way that each depth $d = 1, 2, \ldots, n_Q$ corresponds respectively to each vertex contained in the vertex set V_Q in a sequence of decreasing degree of vertices [19].

The purpose of this was to supplement the consequence of the refinement procedure executed at vertices close to the root vertex in a tree and/or an arbitrary vertex in a graph, considering the fact that a vertex with maximum degree has more adjacent vertices than a vertex with minimum degree [20].

The algorithm uses a vector defined in Equation (2.8) below [19]

$$\vec{H} = \{H_1, \ldots, H_d, \ldots, H_{n_Q}\} \tag{2.8}$$

to record which column is selected at a specified depth. This is to record visited and un-visited vertices so that the algorithm does not visit the same vertex more than once. Note that $H_d = k$ means that the k^{th} column has been selected at depth d. And $H_1 = 0$ means that no column has been selected as yet at depth 1, and $H_1 = 1$ means that the 1^{st} column has been selected at depth 1 [19].

The algorithm uses another vector in Equation (2.9) below [19]

$$\vec{F} = \{F_1, \ldots, F_i, \ldots, F_{n_G}\} \tag{2.9}$$

which works simultaneously with the vector in Equation (2.8), to record which column has been assigned and used at a halfway point of the computation. Since the vector in Equation (2.8) works by deciding which column has been selected at a specified depth, the vector in Equation (2.9) works by deciding whether that selected column has been used for the assignment at a specified depth.

The vector defined in Equation (2.9) is a vector of ones and zeros; these kind of vectors are called binary vectors. The entry F_i in the vector $\vec{F}$ signifies that the i^{th} column has been used if the entry F_i is equal to one, i.e., $F_i = 1$ [19]. However, if the entry F_i is equal to zero then this implies that the i^{th} column has not been used as yet, i.e., $F_i = 0$.

These vectors assists us in marking visited and un-visited columns in the process of assigning vertices that could lead to a one-to-one correspondence in order to construct possible candidate permutation matrices, that could possibly lead to subgraph isomorphism detection. The algorithm traverses and searches through the initialized candidate permutation matrix P^0 just as in the Depth First Tree Search algorithm.

Below is the original Ullmann simple enumeration algorithm which was formulated by Ullmann in 1976 [19], and along side with a flow chart of the original Ullmann simple enumeration algorithm formulated by Mitchell in [18]. Please note that we manipulated the flow chart and added our slight improvement to the algorithm to detect both exact and inexact matches at the same time.

Listing 2.4: **Original Ullmann simple enumeration algorithm, [19].**

```
(1)  Initializing  : M = P^{0}, d = 1;  H_{1} = 0;

     for  all  i  = 1,  ..  ,  n_{Q}, ,  set  F_{i}= 0;

(2)  If  there  does not exist  a  value of  j  such that  the  entry  m_{dj} = 1,
     and F_{j}  = 0  then go to  (7) ;

     P^{d} = M,
     If  depth d = 1  then H_{1} = k, else  k  =  0,

(3)  Advancing : k := k+1,
     If  the  entry  m_{dk} = 0 and or F_{k} = 1 then go to  (3) ;

     for  all  j  not equal to  k  set  the  entry  m_{dj} := 0,

(4)  If  d < n_{Q} then go to (6) ,  else  use the  condition in  equation  (2.12) ,

     and output an isomorphism if found;

(5)  If  there  does not exist  a  j  > k such that  the  entry  m_{dj} = 1
     and F_{j}  = 0  then go to  (7) ;

(6)  Set  H_{d} = k, F_{k} = 1;  then advance d = d+1; go to  (2) ,

(7)  If  d = 1  then terminate the  algorithm,
     F_{k} = 0;  back tracking  d :=  d−1, M = P^{d}, H_{d} = k, go to (5) ,
```

In the original Ullmann simple enumeration algorithm the matrix P^0 is used as the initial candidate matrix. This enumeration algorithm traverses and searches through the initialized matrix in order to assign vertices in V_Q to their candidates in V_G satisfying the one-to-one correspondence condition. This then allows us to construct all candidate permutation matrices that can be verified whether they specify an isomorphism or not.

And to start with the first vertex, we can pick the vertex at depth one, $d = 1$. Note that at depth one the vertices have not been assigned as yet, hence the vector H_1 is equated to zero. Note that the algorithm searches through the initial candidate matrix P^0 and creates other candidate matrices. It then updates the candidate matrices by recursively moving through each depth $d := d + 1$ and back tracks $d := d - 1$.

The algorithm goes through each column entries in the candidate matrices to assign vertices and then back tracks to check if all the vertices have been assigned. This is done by searching for ones and zeros in each column entry of the candidate permutation matrices.

Below is our modified version of a flow diagram [18] for the original Ullmann simple enumeration algorithm.

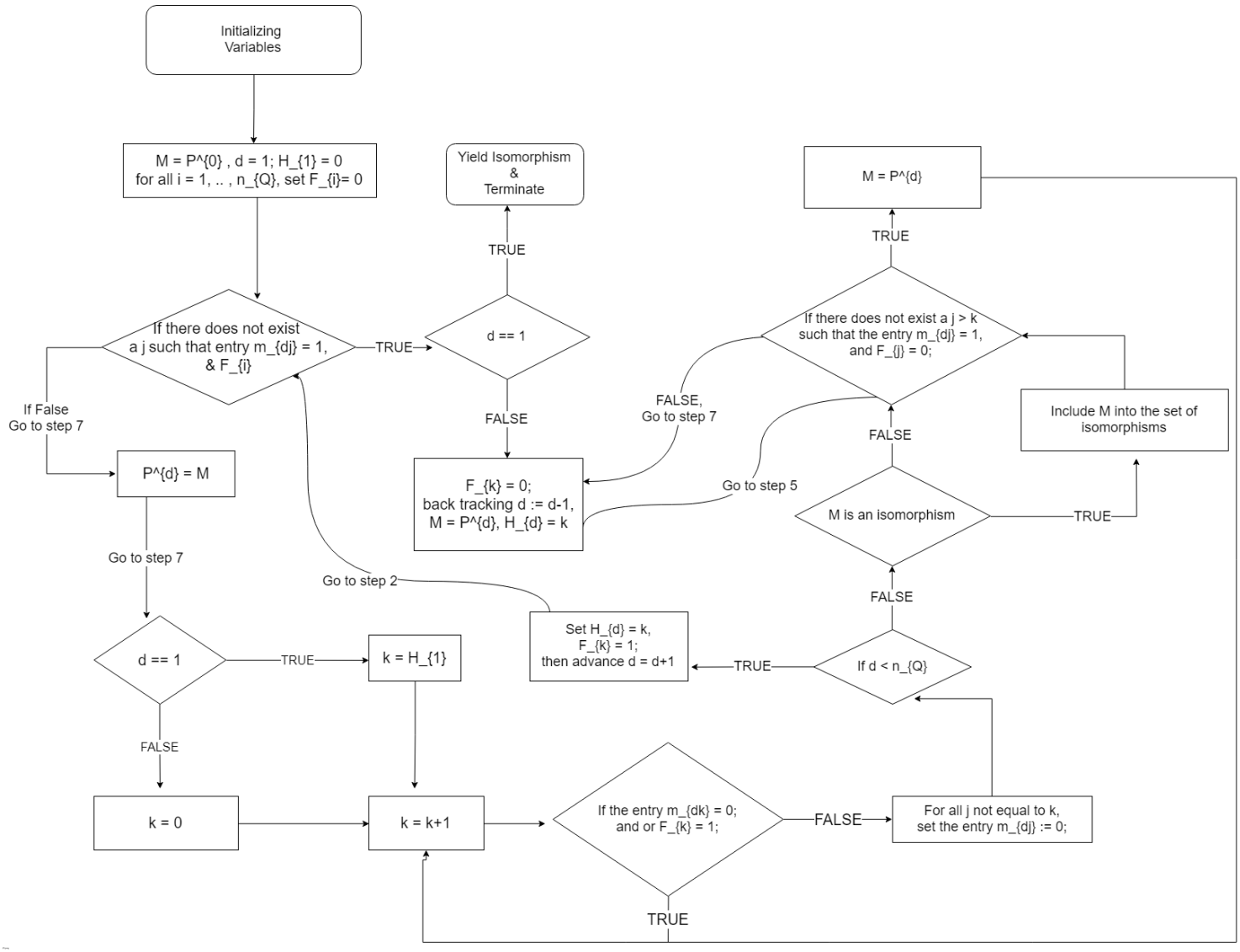


Figure 2.3: Flow Chart of the original Ullmann simple enumeration algorithm

The goal of the Subgraph Isomorphism Problem is to detect all matches (isomorphisms) between a given graph called a pattern graph, denoted by $Q = (V_Q, E_Q)$, and the subgraphs $G_s = (V_s, E_s)$ of a larger graph called the data graph, denoted by $G = (V_G, E_G)$; these matches may be exact or inexact matches. The orders and sizes of the pattern and data graph may be distinct; that is $n_Q \leq n_G$, as mentioned in **Note2**. The pattern graph may be fairly small or equal in comparison to the data graph. The pattern graph has order n_Q and size m_Q; and the data graph is a massive graph with order n_G and size m_G.

Note3: $(P^d)^T$ is called the transposition of the matrix P^d.

The possible candidate permutation matrices contained in the set χ are essential because they are used to permute the rows and columns of the adjacency matrix B_G by multiplying the adjacency matrix B_G from the right with $(P^d)^T$ and from the left with P^d to further construct other matrices whose order and size must be the same as the adjacency matrix A_Q of the pattern graph for detecting both graph and subgraph isomorphism [17].

In constructing other logical representations of G, each possible candidate permutation matrix P^d at depth d is used to permute rows and columns of the adjacency matrix B_G of the large graph G as shown below, just as given in [17],

$$B^d = [b^d_{rc}] = P^d B_G (P^d)^T, \ \ \forall r, c \in \{1, \ldots, n_G\}, \ \ \forall\, d. \tag{2.10}$$

In [17], $P^d = [p^d_{r,c}]$ is defined to be an $n_Q \times n_G$ possible candidate permutation matrix at depth d that is used to permute the adjacency matrix B_G.

Given a massive data graph $G = (V_G, E_G)$ and its adjacency matrix B_G, permuting the rows and columns of B_G using P^d as in the definition above, results into other logical representations of the data graph $G = (V_G, E_G)$ [17].

In Equation (2.10) above, the permutation of rows and columns of the adjacency matrix B_G of the data graph G are performed resulting in other matrices B^d representing the data graph. These shall be called the possible candidate adjacency matrices of the data graph. It is essential to note that if $p^d_{r,c} = 1$ this implies that the c^{th} vertex in the adjacency matrix B_G changes its position to the r^{th} vertex within the candidate adjacency matrix B^d which was constructed by employing Equation (2.10), refer to [19].

Please note that for comparison purposes, the possible candidate adjacency matrices B^d must be of the same dimensions as the adjacency matrix A_Q of the pattern graph $Q = (V_Q, E_Q)$. The dimensions of the possible candidate adjacency matrices B^d are formulated as follows, see [19],

$$(n_Q \times n_G)(n_G \times n_G)(n_G \times n_Q) = n_Q(n_G \times n_G)(n_G \times n_G)n_Q = (n_Q \times n_Q). \tag{2.11}$$

As given by the definition in Equation (2.10), it is essential to mention that given Q and G as large graphs with respective adjacency matrices A_Q and B_G. There exists a subgraph isomorphism between the graph Q and the subgraph G_s of the the data graph G if there exists a possible candidate matrix $P^d = [p^d_{r,c}]$ such that for each $d \in \{1, \ldots, n_Q\}$, $B^d = P^d B_G (P^d)^T$ is

a possible match for the adjacency matrix A_Q of the pattern graph.

Think of the possible candidate permutation matrices contained in χ as bijective functions that map vertices from the vertex set V_Q to the vertex set V_G. It is essential to note that permutations are commonly known to be bijections. Therefore, the question asked by the Subgraph Isomorphism Problem is as close as to finding possible candidate permutation matrices in χ that are used to permute the rows and columns of the adjacency matrix of G as in Equation (2.10).

It is significant to generate all possible candidate permutation matrices $\chi = \{P^0, P^1, P^2, \ . \ . \ .\}$ and thus verify if they indeed specify an isomorphism or not. If an entry $p^d_{r,c}$ within one of the possible candidate matrices in χ is equal to 1 for any r and c, this implies that the c^{th} vertex in B_G corresponds to the r^{th} vertex in the candidate adjacency matrix B^d; otherwise there is no correspondence, i.e., $p^d_{r,c} = 0$ [17].

It is clear that if all row entries and all column entries of the adjacency matrix $A_Q = [a_{rc}]$ correspond to all row entries and column entries of one of the candidate adjacency matrix B^d at depth d, contained in the set of all possible candidate adjacency matrices denoted by B, then the candidate permutation matrix P^d specifies a subgraph isomorphism between the pattern graph Q and the subgraph of the data graph G [17]. That is, if it is true that the condition in Equation (2.12) is satisfied, see below

$$a_{rc} = 1 \implies b^d_{rc} = 1 \ \forall r \in \{1, \ldots, n_Q\}, \ c \in \{1, \ldots, n_Q\}, \tag{2.12}$$

then there is a correspondence [19]. The possible candidate permutation matrices in χ help us to construct all possible candidate adjacency matrices in which each candidate adjacency matrix is compared to the adjacency matrix A_Q to verify if the matrices in χ specify an isomorphism.

The condition in Equation (2.12) allows us to compare the adjacency matrix of the pattern graph to all the possible candidate adjacency matrices contained in B. If Equation (2.12) is satisfied, then the two matrices being compared imply that there is an isomorphism detected, i.e., one of the candidate permutation matrices in χ specifies an isomorphism between the pattern graph Q and the subgraph of the data graph G.

We shall consider the initialized candidate permutation matrix given in Equation (2.7) to illustrate how subgraph isomorphism is detected. We first construct all the possible candidate permutation matrices contained in the set χ and then we permute the rows and columns of the adjacency matrix of G. We shall consider three examples to illustrate how the Ullamnn algorithm works; in the first example we consider Equation (2.7) which deals with adjacency matrices of order 5×5 for the pattern graph Q and an 11×11 adjacency matrix B_G for the data graph G; and the other example deals with adjacency matrices of order 3×3 for the pattern graph Q and 4×4 for the data graph G.

The third example deals with graphs that have the same order, and the same size. In this example we go through an in-depth mathematical explanation to illustrate how the graph and subgraph isomorphism is detected.

In the process of developing an implementation for the Ullmann algorithm, we use Java as the language of preference.

Example 1

In the example below we consider the initial permutation matrix P^0 which is defined in Equation (2.7) for an in-depth analysis of the original Ullmann algorithm. Here, we construct all the possible candidate permutation matrices in the set χ, which are then used to construct all the possible candidate adjacency matrices contained in the set B for the data graph.

We shall then compare these possible candidate adjacency matrices with the adjacency matrix of the pattern graph A_Q. For this example we constructed a java program to construct all these possible candidate permutation matrices along with their corresponding candidate adjacency matrices; refer to the Appendix A for the java program.

Listing 2.5: **Results**

```
run:

The  initialized   candidate permutation matrix
1 1 0 0 0 1 1 0 0 0 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

Level is  Starting
Is  this  row a ID row 0 :  false
Is  this  row a ID row 1 :  false
Is  this  row a ID row 2 :  false
Is  this  row a ID row 3 :  false
Is  this  row a ID row 4 :  false
The row 0 column : -1
The row 1 column : -1
The row 2 column : -1
The row 3 column : -1
The row 4 column : -1
1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

0 1 0 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

0 0 0 0 0 0 1 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 1 0 0 0 0
```

```
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

********** this is level 1**********
Is this row a ID row 0 : true
Is this row a ID row 1 : false
Is this row a ID row 2 : false
Is this row a ID row 3 : false
Is this row a ID row 4 : false
The row 0 column : 0
The row 1 column : -1
The row 2 column : -1
The row 3 column : -1
The row 4 column : -1
1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

1 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
```

```
1 1 1 1 1 1 1 1 1 1 1

1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

Is this row a ID row 0 : true
Is this row a ID row 1 : false
Is this row a ID row 2 : false
Is this row a ID row 3 : false
Is this row a ID row 4 : false
The row 0 column : 1
The row 1 column : −1
The row 2 column : −1
The row 3 column : −1
The row 4 column : −1
0 1 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

...

**********This continues until all levels are explored**********

**** By using the candidate permutation matrices we construct candidate adjacency matrices****

Is this row a ID row 0 : true
Is this row a ID row 1 : true
```

```
Is this row a ID row 2 : true
Is this row a ID row 3 : true
Is this row a ID row 4 : false
The row 0 column : 10
The row 1 column : 9
The row 2 column : 8
The row 3 column : 5
The row 4 column : −1
Is this row a ID row 0 : true
Is this row a ID row 1 : true
Is this row a ID row 2 : true
Is this row a ID row 3 : true
Is this row a ID row 4 : false
The row 0 column : 10
The row 1 column : 9
The row 2 column : 8
The row 3 column : 6
The row 4 column : −1
Is this row a ID row 0 : true
Is this row a ID row 1 : true
Is this row a ID row 2 : true
Is this row a ID row 3 : true
Is this row a ID row 4 : false
The row 0 column : 10
The row 1 column : 9
The row 2 column : 8
The row 3 column : 7
The row 4 column : −1
size is2520
********** this is level 4**********
Is this row a ID row 0 : true
Is this row a ID row 1 : true
Is this row a ID row 2 : true
Is this row a ID row 3 : true
Is this row a ID row 4 : false
The row 0 column : 0
The row 1 column : 1
The row 2 column : 2
The row 3 column : 3
The row 4 column : −1
This is the candidate permutation matrix
1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
This is the resulting candidate adjacency matrix
0 1 1 0 0
1 0 0 1 0
1 0 0 0 0
0 1 0 0 0
0 0 0 0 0
END
Partial Vertex Similarities

This is the candidate permutation matrix
```

1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
This is the resulting candidate adjacency matrix
0 1 1 0 0
1 0 0 1 0
1 0 0 0 1
0 1 0 0 0
0 0 1 0 0
END
Partial Vertex Similarities

This is the candidate permutation matrix
1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0
This is the resulting candidate adjacency matrix
0 1 1 0 0
1 0 0 1 0
1 0 0 0 0
0 1 0 0 1
0 0 0 1 0
END
Partial Vertex Similarities

This is the candidate permutation matrix
1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0
This is the resulting candidate adjacency matrix
0 1 1 0 0
1 0 0 1 0
1 0 0 0 0
0 1 0 0 0
0 0 0 0 0
END
Partial Vertex Similarities

This is the candidate permutation matrix
1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0
This is the resulting candidate adjacency matrix
0 1 1 0 0
1 0 0 1 0
1 0 0 0 0
0 1 0 0 0
0 0 0 0 0

```
END
 Partial  Vertex  Similarities

This  is  the  candidate permutation matrix
1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0
This  is  the  resulting  candidate adjacency matrix
0 1 1 0 0
1 0 0 1 1
1 0 0 0 0
0 1 0 0 0
0 1 0 0 0
END
 Partial  Vertex  Similarities

This  is  the  candidate permutation matrix
1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
This  is  the  resulting  candidate adjacency matrix
0 1 1 0 1
1 0 0 1 1
1 0 0 0 0
0 1 0 0 0
1 1 0 0 0
END
 Partial  Vertex  Similarities

**********This continues until all  candidate adjacency matrices are constructed**********

*********There are 17640 possible candidate permutation matrices which are constructed*********

*********We expect 17640 possible candidate adjacency matrices to be constructed*********

BUILD SUCCESSFUL (total time: 19 seconds)
```

Example 2

Let us consider two graphs, namely the pattern graph $Q = (V_Q, E_Q)$ and the data graph $G = (V_G, E_G)$. In this example the vertex set of the pattern graph consists of three vertices and the vertex set of the data graph consists of four vertices. We consider the two given graphs along with their corresponding adjacency matrices, i.e., A_Q is the adjacency matrix for the pattern graph and B_G is the adjacency matrix for the data graph:

$$A_Q = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix};$$

$$B_G = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

The initialized candidate permutation matrix is given by P^0, shown below.

$$P^0 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Represented in Listing 2.6 are the results achieved after running the main java program included in Appendix B, we use the algorithmic concepts derived by Ullmann in his original algorithm [19] to show how his algorithm works and how it creates all the possible candidate permutation matrices; in turn creating its respective candidate adjacency matrices. The Java programs included in Appendix B construct a collection of all the possible candidate permutation matrices required to develop all the respective candidate adjacency matrices for comparison purposes.

All that's left to compute, is the adjacency matrix comparison by implementing a matrix permutation just as in Equation (2.10), [19]. After the computation is done, we get all the possible candidate permutation matrices along with their respective candidate adjacency matrices.

The adjacency matrix A_Q of the pattern graph is then compared to all the resulting candidate adjacency matrices in Listing 2.6; and it has come to our attention that all the resulting candidate permutation matrices specify an isomorphism since their corresponding candidate adjacency matrices in Listing 2.6 match the adjacency matrix A_Q of the pattern graph.

Given the adjacency matrix of the pattern graph,

$$A_Q = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix},$$

notice that all the generated candidate adjacency matrices match the adjacency matrix of the pattern graph.

Listing 2.6: **Results**

```
run:

The initialized candidate permutation matrix
1 0 1 1
1 1 1 1
0 1 0 0

Level is starting
Is this row a ID row 0 : false
Is this row a ID row 1 : false
Is this row a ID row 2 : true
The row 0 column : -1
The row 1 column : -1
The row 2 column : 1
1 0 0 0
1 1 1 1
0 1 0 0

0 0 1 0
1 1 1 1
0 1 0 0

0 0 0 1
1 1 1 1
0 1 0 0


************************ this is level 1**********************

Is this row a ID row 0 : true
Is this row a ID row 1 : false
Is this row a ID row 2 : true
The row 0 column : 0
The row 1 column : -1
The row 2 column : 1
This is the candidate permutation matrix
1 0 0 0
0 0 1 0
0 1 0 0

This is the resulting candidate adjacency matrix
0 0 1
0 0 1
1 1 0
END
Exact Match

Is this row a ID row 0 : true
Is this row a ID row 1 : false
Is this row a ID row 2 : true
The row 0 column : 0
The row 1 column : -1
The row 2 column : 1
This is the candidate permutation matrix
1 0 0 0
```

0 0 0 1
0 1 0 0

This is the resulting candidate adjacency matrix
0 0 1
0 0 1
1 1 0
END
Exact Match

Is this row a ID row 0 : true
Is this row a ID row 1 : false
Is this row a ID row 2 : true
The row 0 column : 2
The row 1 column : −1
The row 2 column : 1
This is the candidate permutation matrix
0 0 1 0
1 0 0 0
0 1 0 0

This is the resulting candidate adjacency matrix
0 0 1
0 0 1
1 1 0
END
Exact Match

Is this row a ID row 0 : true
Is this row a ID row 1 : false
Is this row a ID row 2 : true
The row 0 column : 2
The row 1 column : −1
The row 2 column : 1
This is the candidate permutation matrix
0 0 1 0
0 0 0 1
0 1 0 0

This is the resulting candidate adjacency matrix
0 0 1
0 0 1
1 1 0
END
Exact Match

Is this row a ID row 0 : true
Is this row a ID row 1 : false
Is this row a ID row 2 : true
The row 0 column : 3
The row 1 column : −1
The row 2 column : 1
This is the candidate permutation matrix
0 0 0 1
1 0 0 0
0 1 0 0

```
This is the resulting candidate adjacency matrix
0 0 1
0 0 1
1 1 0
END
Exact Match

Is this row a ID row 0 : true
Is this row a ID row 1 : true
Is this row a ID row 2 : true
The row 0 column : 3
The row 1 column : 2
The row 2 column : 1
This is the candidate permutation matrix
0 0 0 1
0 0 1 0
0 1 0 0

This is the resulting candidate adjacency matrix
0 0 1
0 0 1
1 1 0
END
Exact Match


There are 6 candidate adjacency matrices

BUILD SUCCESSFUL (total time: 0 seconds)
```

It is of great significance to note that the implementation works perfectly for a small data set; however, it tends to get slower as the number of vertices and edges increases. This is due to the fact that the Ullmann algorithm uses the brute-force enumeration procedure, which rapidly terminates to FAIL as the data set increases.

Example 3

Let us now consider an example for graphs of the same order and of the same size, we can construct the possible candidate matrices by using the definition of permutation. Note that the method involves permuting vertices from vertex set V_Q to those in vertex set V_G. Below is a definition we used to construct P^d.

Two graphs Q and G are said to be isomorphic, i.e., $G \cong Q$, if and only if there exists a bijection $\delta : V_Q \mapsto V_G$ such that $(\delta(u), \delta(v)) \in E_G$ for vertices $u, v \in V_Q$, if and only if $(u, v) \in E_Q$.

Finding the bijective map δ is as close to finding the permutation of rows and columns of the adjacency matrix B_G. The permutation δ is used as reference for constructing the candidate permutation matrices P^d. See example below.

Given two graphs G_1 and G_2 as shown below:

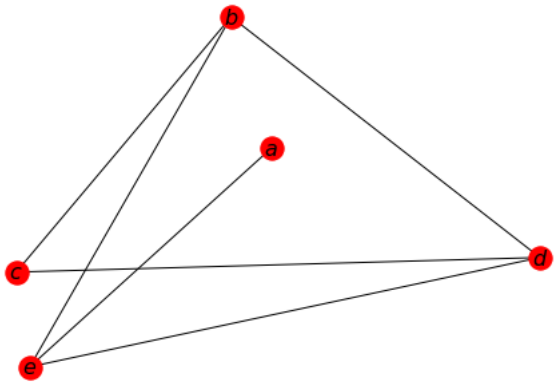


Figure 2.4: Graph G_1

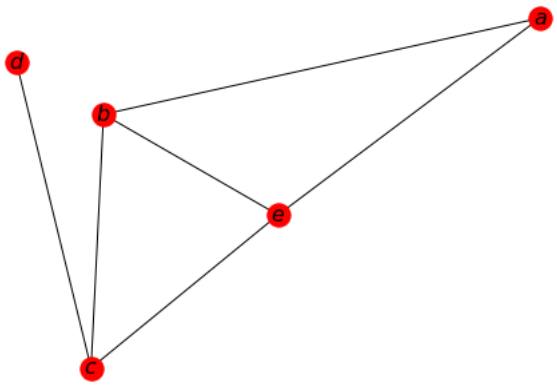
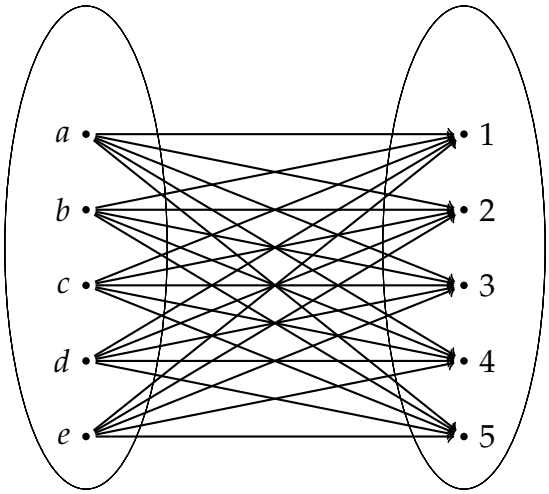


Figure 2.5: Graph G_2

The most important question we want to answer is whether these graphs are isomorphic, that

is, $G_2 \cong G_1$?. In the process of answering this question we are required to find any possible candidate bijection $\delta : V_{G_1} \mapsto V_{G_2}$. Note that this possible bijection can be thought of as the possible candidate permutation, which in turn is used to construct its respective candidate permutation matrix. Hence, this candidate permutation matrix shall be used to permute the adjacency matrix of the graph G_2.

This example consists of $5! = 5 \times 4 \times 3 \times 2 \times 1$ different types of permutations δ; this is the possible number of ways that each vertex in V_{G_1} can be mapped to each vertex in V_{G_2}, which implies that each and every vertex in the vertex set V_{G_1} can be mapped to each and every vertex in the vertex set V_{G_2}, refer to [11].



In constructing the candidate permutation matrices we shall have the following permutations as reference:

$$\delta^1 = \begin{pmatrix} a & b & c & d & e \\ a & b & c & d & e \end{pmatrix} \tag{2.13}$$

$$\delta^2 = \begin{pmatrix} a & b & c & d & e \\ b & c & d & e & a \end{pmatrix} \tag{2.14}$$

$$\delta^3 = \begin{pmatrix} a & b & c & d & e \\ c & d & e & a & b \end{pmatrix} \tag{2.15}$$

$$\delta^4 = \begin{pmatrix} a & b & c & d & e \\ d & e & a & b & c \end{pmatrix} \tag{2.16}$$

$$\delta^5 = \begin{pmatrix} a & b & c & d & e \\ e & a & b & c & d \end{pmatrix} \tag{2.17}$$

$$\delta^6 = \begin{pmatrix} a & b & c & d & e \\ e & d & a & b & c \end{pmatrix} \tag{2.18}$$

$$etc \ldots$$

By the use of Cauchy's two line notation for a permutation we get some of the possible permutations of the vertices in the vertex set V_{G_1} to the vertices in V_{G_2}, as shown above. In general, we may denote the k^{th} possible candidate permutation by

$$\delta^k = \begin{pmatrix} x_1 & x_2 & \ldots & x_n \\ \delta(x_1) & \delta(x_2) & \ldots & \delta(x_n) \end{pmatrix}, \tag{2.19}$$

where $x_1, x_2, \ldots, x_n$ are the vertices. And those in the second line $\delta(x_1), \delta(x_2), \ldots, \delta(x_n)$ are the possible arrangements of those in the first line [11].

In the process of constructing the candidate permutation matrices, we pick a mapping from all possible mappings δ^k for all k, and then verify if it is a bijection. Suppose we have the following as the possible candidate permutation,

$$\delta^6 = \begin{pmatrix} a & b & c & d & e \\ e & d & a & b & c \end{pmatrix}.$$

This means that $a \in V_{G_1}$ is mapped to $e \in V_{G_2}$, $b \in V_{G_1}$ is mapped to $d \in V_{G_2}$, $c \in V_{G_1}$ is mapped to $a \in V_{G_2}$, $d \in V_{G_1}$ is mapped to $b \in V_{G_2}$, and vertex $e \in V_{G_1}$ is mapped to the vertex $e \in V_{G_2}$ [22].

By using the result in Equation (2.18) we are able to construct its respective candidate permutation matrix, see below:

$$P^6 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}. \tag{2.20}$$

The transposition of the candidate permutation matrix is given in Equation (2.21) below :

$$(P^6)^T = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{2.21}$$

The candidate permutation matrix in Equation (2.20) is constructed from its respective map or permutation defined in Equation (2.19). We are now required to verify whether δ^6 is indeed a

bijective map from the vertex set V_{G_1} to the vertex set V_{G_2} that preserves adjacency and non-adjacency. Note that if δ^6 is a bijection, this then implies that the candidate permutation matrix in Equation (2.20) is also a suitable permutation matrix to generate its respective candidate adjacency matrix B^6 by satisfying the condition in Equation (2.10) [19].

In order to verify that the candidate permutation matrix P^6 specifies an isomorphism we need to show that δ^6 is indeed an isomorphism; we need to scan each pair of vertices in the vertex set V_{G_1}, and prove that if they were adjacent in the graph G_1 then after the mapping, they get mapped onto a pair of vertices that are adjacent in the graph G_2. Likewise, if these vertices were not adjacent in the graph G_1 then mapping them yields a pair of vertices which are not adjacent in the graph G_2 [22].

In demonstrating adjacency and non-adjacency we pick a pair of vertices, say $(a, e) \in E_{G_1}$ and notice that mapping these vertices yields a pair of adjacent vertices $(\delta^6(a), \delta^6(e)) = (e, c) \in E_{G_2}$. Likewise, if we pick a pair of vertices which are not adjacent, say $(a, b) \notin E_{G_1}$. This implies that their mapping yields a pair of vertices, $(\delta^6(a), \delta^6(b)) = (e, d) \notin E_{G_2}$, which are not adjacent in G_2. This allows us to conclude that δ^6 is an isomorphism, by scanning through every vertex in the graph G_1 comparing them to those in the graph G_2.

However, note that to verify isomorphism in terms of the adjacency matrices for graphs G_1 and G_2, we shall construct the adjacency matrices of the two graphs and then use the condidate permutation matrix in Equation (2.20) to permute the adjacency matrix of the graph G_2; hence satisfying the condition in Equation (2.10).

That is, for the given example we have the adjacency matrices A_{G_1} and B_{G_2} which are given below:

$$A_{G_1} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}; \tag{2.22}$$

$$B_{G_2} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \tag{2.23}$$

So according to the condition in Equation (2.10), we are able to construct another candidate adjacency representation of G_2 by permuting the rows and columns of B_{G_2}. See below, as in [19], that

$$B^6 = P^6 B_{G_2} (P^6)^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} = A_{G_1} .$$

It is clear that the two graphs G_1 and G_2 are isomorphic, this implies that the candidate permutation matrix P^6 preserves adjacency and non-adjacency in the given graphs; hence specifying isomorphism between the two graphs.

2.2 Implementation of the Refinement Procedure

J. R. Ullmann further develops his original algorithm for subgraph isomorphism [19]. He implements a procedure called the refinement procedure to minimize the computational processing costs. This refinement procedure deletes some of the ones from the possible candidate matrices, which in turn deletes some of the successor vertices.

The refinement procedure considers the candidate permutation matrices that are related with any given non-terminal vertices in order to eliminate some of the ones and thus eliminating successor vertices. It is significant to note that any subgraph isomorphism corresponds to a specific candidate permutation matrix just as stated by Ullmann in [19]. The subgraph isomorphism is an isomorphishm under a specified candidate permutation matrix which is thought of as a bijective function, and should be associated with terminal vertices at depth $d = n_Q$ and successor vertices [19].

The zero entries $p^d_{r,c} = 0$ within the candidate permutation matrices imply that there is no correspondence between vertices in the vertex set V_Q and vertices in the vertex set V_G. All the terminal vertices that correspond to distinct candidate permutation matrices, say P^d, are used as reference for the refinement procedure. The refinement procedure prunes out all the candidate vertices, say $v_c \in C^d(v_r)$, that have less adjacent vertices than v_r [16].

If the entry $p^d_{r,c} = 0$ is obtained in a candidate permutation matrix P^d for subgraph isomorphism, then if there is an entry in P^{d+1} with $p^{d+1}_{r,c} = 1$ corresponding to vertices that have smaller degree, we can change $p^{d+1}_{r,c} = 1$ to $p^{d+1}_{r,c} = 0$ without compromising subgraph isomorphism under a specified candidate permutation matrix, P^d [19].

The significant part of the original Ullmann's algorithm is the refinement of the candidate permutation matrices that are compatible for isomorphism. Ullmann formulated a condition that is mandatory to hold for an entry $p^d_{r,c}$ to be one for any subgraph isomorphism under P^d. Suppose that this condition does not hold and that the entry $p^d_{r,c} = 1$, then the refinement process turns the entry $p^d_{r,c} = 1$ to $p^d_{r,c} = 0$ [19].

Ullmann constructed the condition as follows, let v_r be an r^{th} vertex in the vertex set V_Q of the pattern graph; similarly, let v_c be a c^{th} vertex in the vertex set V_G of the data graph. Let the set of all vertices that are adjacent to $v_r \in V_Q$ be denoted by $\{v_1, \ldots, v_i, \ldots, v_{n_Q}\} \in V_Q$. Recall that P^d is a possible candidate permutation matrix associated with any specified isomorphism between two given graphs [19].

By the definition of the subgraph isomorphism, it is significant to note that if a vertex $v_r \in V_Q$ in the pattern graph matches a vertex $v_c \in V_G$ in the data graph, then there should be another candidate vertex $v_j \in V_G$ in the data graph adjacent to the vertex $v_c \in V_G$, such that this candidate vertex $v_j \in V_G$ matches the vertex $v_i \in V_Q$, for every $i = \{1, \ldots, n_Q\}$, [19].

If an isomorphism is specified, an entry of the candidate permutation matrix P^d corresponding to a relation $\{v_i, v_j\}$ is equal to 1 if the vertex $v_j \in V_G$ is a compatible candidate match of

the vertex $v_i \in V_Q$, [19].

Suppose that the vertex $v_r \in V_Q$ matches vertex $v_c \in V_G$ in any subgraph isomorphism test under the permutation matrix P^d, then the entry element in the permutation matrix P^d corresponding to some relation $\{v_i, v_j\}$ is equal to 1 for every $i = \{1, \ldots, n_Q\}$ such that the vertex $v_j \in V_G$ is adjacent to the vertex $v_c \in V_G$.

J. R. Ullmann constructed this mandatory condition in [19] as follows, if the vertex $v_r \in V_Q$ matches the vertex $v_c \in V_G$ in any subgraph isomorphism specified under the candidate permutation matrix P^d, then

$$(\forall i)((a_{ri} = 1) \implies (\exists j)(p^d_{ij} \times b_{jc} = 1)) \ \forall i \in [1, n_Q], \ j \in [1, n_Q]. \tag{2.24}$$

The Ullmann algorithm implements the refinement procedure by purely examining every 1 entry in the candidate permutation matrix P^d to detect if the condition in Equation (2.24) holds. If the condition in Equation (2.24) does not hold for any entry $p^d_{rc} = 1$ in P^d, then the entry $p^d_{rc} = 1$ is changed to a zero entry, i.e., $p^d_{rc} = 0$.

This could lead to the condition in Equation (2.24) not to hold for other 1 entries in the candidate permutation matrix P^d so that additional changes can be implemented and so forth, [20]. The Ullmann algorithm executes this refinement process repeatedly, up until all the 1 entries in the matrix P^d have been examined and none of them can be changed to 0.

Above all, the main objective of the refinement procedure is to modify all the 1 entries in the matrix P^d to be 0 entries. It is also essential to note that the this procedure may leave the candidate permutation matrix unaffected provided that the matrix P^d specifies a subgraph isomorphism. If the candidate permutation matrix is left unchanged by the refinement process then this would imply that the condition in Equation (2.24) holds for every 1 entry in the matrix P^d, [3].

Thus implying that the matrix P^d enumerates a one-to-one correspondence between the vertex set V_Q and the vertex set V_G so that if a pair of distinct vertices $v_r, v_i \in V_Q$ are adjacent to each other, then their corresponding pair of vertices $v_c, v_j \in V_G$ must also be adjacent to each other, [19].

The condition in Equation (2.24) is clearly an improvement to the condition in Equation (2.12). The refinement procedure aims to satisfy the condition in Equation (2.24) when checking for a subgraph isomorphism instead of satisfying the condition in Equation (2.12). The refinement procedure regularly inspects if the rows of the candidate permutation matrix contains one or not; and if there are no ones in any of the rows then the procedure terminates since there are no more ones to proceed with the refinement procedure, [19].

By implementing the refinement procedure, the original Ullmann simple enumeration algorithm in Listing 2.4 is improved to apply and satisfy the condition in Equation (2.24), see Listing 2.7 in the page that follows.

Listing 2.7: **Algorithm using the refinement procedure, [19].**

```
(1)  Initializing  : M = P^{0}, d = 1;  H_{1} = 0;

     for  all  i  =  1,  ..  ,  n_{Q}, ,  set  F_{i}= 0;

     refine  M, if  exit  FAIL then terminate algorithm;

(2)  If  there  does not exist  a  value of  j  such that  the  entry  m_{dj} = 1;
     and F_{j}  = 0  then go to  (7) ;

     P^{d} = M,
     If  depth d = 1  then H_{1} = k, else  k  =  0;

(3)  Advancing : k := k+1,
     If  the  entry  m_{dk} = 0 and or F_{k} = 1 then go to  (3) ;

     for  all  j  not equal to  k  set  the  entry  m_{dj} := 0;

     refine  M, if  exit  FAIL then go to  (5) ;

(4)  If  d  <  n_{Q} then go to (6) ,  else  use the  condition in  equation (13) ;

     and output an isomorphism if found;

(5)  If  there  does not exist  a  j  >  k such that  the  entry  m_{dj} = 1
     and F_{j}  = 0  then go to  (7) ;

(6)  Set  H_{d} = k, F_{k} = 1;  then advance d = d+1; go to  (2) ;

(7)  If  d  =  1  then terminate the  algorithm;

     F_{k} = 0;  back tracking  d  :=  d−1, M = P^{d}, H_{d} = k, go to (5) ;
```

The Ullmann algorithm was employed to detect subgraph isomorphism and this occurs inside a finite period of time. To estimate the computational processing period that the algorithm takes, Ullmann adopted an experimental procedure for subgraph isomorphism detection.

2.3 Improvements to Ullmann's algorithm

In [4], Čibej and Mihelič formulated a function $P^{d+1} = refine(r, c, P^d)$ which examines all the entries in the candidate matrices. The objective of this refinement procedure is to minimize the number of ones in each candidate matrix by analyzing the neighborhoods of each compatible pair of vertices [4].

The refinement procedure satisfies a condition that states:

"For a pair of vertices v_r and v_c to remain compatible vertices, every vertex in the neighborhood of vertex v_r should occupy more than one compatible vertex in the neighborhood of vertex v_c. That is to say, $p^d_{r,c} = 1 \iff \forall u_i \in N_Q(v_r)\ \exists u_j \in N_G(v_c)$ such that $p^d_{i,j} = 1$", [4].

The refinement procedure satisfies this condition for every compatible pair of vertices. It is important to note that every pair of vertices that do not satisfy this condition also stay incompatible. The refinement procedure changes the one entries of a compatibility matrix to zero until no more ones can be changed to zero [3].

According to Čibej and Mihelič the refinement procedure is an utterly costly process. However, due to its high computational processing costs, up to this modern day, a collection of distinct improvements were developed in [3] to enhance the computational processing costs of the Ullmann algorithm.

Čibej and Mihelič outlined the main points of the improvements in [4] and named it the UI (short for Ullmann Improved) [4].

Čibej and Mihelič Improvements to Ullmann algorithm entail the following:

- **Filtering development :**

  If we assign a vertex v_r to another vertex v_c, these vertices become incompatible with every other vertex in the graph. That is, there is a one-to-one correspondence that is satisfied. It is essential to note that vertices contained in the neighborhood $N_Q(v_r)$ are always compatible to vertices contained in the neighborhood $N_G(v_c)$, but incompatible with vertices that are not contained in $N_G(v_c)$, [3].

- **Refinement development :**

  The entire refinement procedure changes all the entries of ones in the candidate matrix to entries of zeros until no more changes can be done [4]. The improved refinement has a partial refinement step, which examines only the final changes made in the candidate matrices, hence improving the algorithms computational costs.

- **Minimizing Storage requirements:**

  The improvements uses historical changes in the compatibility candidate matrices which

minimizes storage from $O(n_Q^3)$ to $O(n_Q^2)$, [4].

In the UI (Ullmann Improved algorithm), Čibej and Mihelič consider a search strategy which is basically the vertex order; it is used to sort vertices in the pattern graph which happen to be less compatible with any vertex in the data graph. The search strategy helps with pruning and thus eliminating the most branches, and minimizing the storage requirements.

Graph theory makes available to us a great amount of features that can be used in sorting vertices. A well established sorting algorithm is made up of two standard principles that the search strategy uses to iteratively pick the next vertex [3].

1. The primary principle constructs a set of candidate vertices, just as in Equation (2.4).
2. The secondary principle is used to pick a node from the candidate set in Equation (2.4).

The above mentioned principles are used interchangeably, to a point where every vertex contained in the pattern graph are in the search strategy. While implementing these principles, a variety of essential features of graphs are considered. The main objective for employing the primary principle is to choose a set of candidate vertices, from which only one vertex shall be picked just as described in the secondary principle [3].

Some of the features that are taken into consideration for the primary principle are **Depth**, **Maximum degree/ subdegree**, **Neighborhood**, and **All vertices**.

"Depth" is one of the features that is considered when constructing the set of all candidate vertices based on the primary principle. In this case the candidate set is the whole level of a depth-first search and/or a breadth-first search [3]. Here, the algorithm begins at a randomly chosen vertex and arrange each vertex at distance one, and then every vertex at distance two, and so on [4].

In the case of the **"Maximum subdegree"**, Čibej and Mihelič defined the vertex subdegree as the number of edges from a vertex in the vertex set of the data graph to the vertices in the vertex set of the pattern graph; Čibej and Mihelič denoted this by the cardinality of the set of already visited vertices [3]

$$D_{V_{is}(v)} = |\{u \in V_{is} | (v,u) \in E_Q\}|. \tag{2.25}$$

It is important to note that the set of all candidate vertices is the set of all vertices containing the maximum subdegree. Čibej and Mihelič denoted the set of all vertices that have maximum subdegree to be M_{deg}, [3].

The primary principle is implemented to select the neighbors of the visited vertices contained in the candidate set V_{is}. This is considered to be one of the features employed by the primary principle, this is called the **"Neighborhood"** [4]. As explained by Čibej and Mihelič, at the start of the algorithm the set of all candidate vertices V_{is} is an empty set, that is to say that each vertex in the pattern graph are candidate vertices for selection based on the secondary principle [3]. This then implies that according to the secondary principle an initial vertex is selected and then the arrangement develops from that vertex.

In the case where each and every vertex is considered a candidate vertex for selection, the

primary principle is not used. However, the secondary principle is considered the only criterion for arranging vertices; the feature considered here is referred to as **"All vertices"** [3].

Below are some of the possible features employed by the secondary principle to involve some instinctive procedures that rapidly prunes the search graph or search tree [3].

- Random
- Degree
- Open-neighborhood clustering
- Closed-neighborhood clustering
- Eccentricity

For the purpose of differentiation, Čibej and Mihelič employ a feature called **"Random"** to randomly sample candidate vertices from the set of vertices with maximum subdegree, M_{deg}, based on the secondary principle.

In [19], J. R. Ullmann presented a feature for arranging the vertices in the pattern graph using their descending degree. With that being said in [4], Čibej and Mihelič also employ ordering in such a way that every vertex that has the highest degree is commonly mapped to a small number of vertices in the data graph, in turn minimizing the search space.

"Open and Closed-neighborhood clustering" are the other two essential features considered in [3] for arranging the vertices in the pattern graph. Open-neighborhood clustering is referred to as the amount of edges a vertex has in its neighborhood. By selecting vertices that have the higher number of edges in the neighborhood further minimizes the search space, since the higher the number of edges in the neighborhood leads to greater constraints in the possible candidate matrices [4].

Similarly, the closed-neighborhood clustering is employed just as the open-neighborhood clustering; however, the closed-neighborhood clustering evaluates all the edges in the neighborhood.

"Eccentricity" of a graph vertex is the maximum distance from a vertex in the graph to any other vertex in that graph. Eccentricity is one of the features used and is of great importance since it also helps in minimizing the search space. The smaller the eccentricity of a vertex, the more compatible the vertex is; this feature allows the search strategy to begin the search at the center of the graph leading to the reduction of the search space since it considers vertices with smaller eccentricity [3].

Chapter 3

3.1 VF2 Algorithm

This section of the dissertation analyzes another significant algorithm for checking both the graph and subgraph isomorphism problems presented by L.P. Cordella and et. al. in [6]; this proposed matching algorithm is called the VF2 algorithm. The main objective behind this algorithm is established on the evaluation of the vertex set of vertices adjacent to the vertices that are already examined in the partial mapping between vertices of two given graphs. L.P. Cordella et al. designed an algorithm called the VF algorithm, then reconstructed and improved it to form the VF2 algorithm which is a more advanced and efficient algorithm for the graph and subgraph isomorphism detection.

According to Lixin Fu and Surya Prakash in [10]; the VF2 algorithm is an exceptional development over the Ullmann algorithm and we shall put a lot of focus on understanding the foundation of this algorithm.

In this section we shall discuss in detail the framework of the VF2 algorithm and thus examine the crucial improvements that were introduced to immensely minimize storage requirements. The VF2 algorithm was designed for working with extremely large graphs (over 3000 vertices and edges), hence why it employs a collection of sensible rules which help us to exceptionally minimize the computational processing costs for graph and subgraph isomorphism verification. The fundamental idea behind these feasible rules is that they permit us to trim out the search space, so as to achieve an effective matching procedure.

The isomorphism procedure between two given graphs, denoted by $G = (V_G, E_G)$ and $Q = (V_Q, E_Q)$ is composed of formulating a relation function $R \subseteq V_Q \times V_G$ that relates vertices contained in the vertex set V_Q of the graph Q with the vertices contained in the vertex set V_G of the graph G and vice versa [21]. Generally, the relation function R is defined to be a collection of pairs of vertices (v, u) with vertices $v \in V_Q$ and $u \in V_G$. Each of the pairs denote a correspondence relation (also known as a mapping) of vertex $v \in V_Q$ onto a vertex $u \in V_G$ [6].

The relation function $R \subseteq V_Q \times V_G$ specifies an isomorphism between the two given graphs Q and G if and only if it describes a one-to-one correspondence between the vertex set V_Q onto the vertex set V_G, and if it preserves the adjacency and non-adjacency of vertices of the two given graphs. The relation R specifies a subgraph isomorphism if and only if it specifies an

isomorphism between the graph $Q = (V_Q, E_Q)$ and a subgraph $G_s = (V_s, E_s)$ of a further given graph $G = (V_G, E_G)$.

The strategy of detecting an exact relation function $R \subseteq V_Q \times V_G$ is solved by adopting the notion of a state space representation (SSR for short) which is briefly discussed in [7]. It is essential to note that every given state i contained in the state space S, represents a partial mapping solution $R(i)$ between the vertex sets V_Q and V_G of the two given graphs with $R(i) \subseteq R$, that is to say, it includes only a subspace of the entire vertex relations in the mapping $R \subseteq V_Q \times V_G$, [6].

Each and every partial mapping solution $R(i) \equiv V_{Q_s}(i) \times V_{G_s}(i)$ is made up of a set of ordered pairs of vertices (v, u), for each vertex $v \in V_{Q_s}(i) \subseteq V_Q$ and $u \in V_{G_s}(i) \subseteq V_G$. Having said that, we let $Q_s(i)$ represent the subgraph of Q and $G_s(i)$ the subgraph of G, by picking only the vertices contained in the partial mapping solution and their corresponding edges which link them. The subgraph $Q_s(i)$ contains a vertex set denoted by $V_{Q_s}(i)$ and an edge set $E_{Q_s}(i)$; similarly, the subgraph $G_s(i)$ contains a vertex set denoted by $V_{G_s}(i)$ and an edge set $E_{G_s}(i)$, [14].

As mentioned earlier, the exact matching procedure is based on an in-depth search within the state space S in such a way that the algorithm begins by selecting an initial state i_0 where the mapping function yields an empty relation $R(i_0) = \emptyset$. The algorithm continues the search inside the state space S for at least one target state i_t such that the mapping function $R(i_t)$ is equivalent to $R \subseteq V_Q \times V_G$; that is to say $R(i_t) \equiv R$, [1].

As stated by L.P. Cordella et al. advancing from a universal state i to a successor state $i' = i \cup (v, u)$ indicates a process of adding new candidate pairs of corresponding vertices (v, u) such that $v, u \notin R(i)$; the state progression from the universal state i to its successor state i' specifies the addition of a vertex v to the subgraph $Q_s(i)$ of Q, and addition of a vertex u to the subgraph $G_s(i)$ of G. It is clear that every successor state i' is developed by using the universal state i as its parent state [1].

All these possible candidate states could be constructed by the use of a brute-force enumeration procedure, and therefore assisting us in generating all the possible target states; supposing they exist. However, the brute-force enumeration procedure rapidly fails and leads to a factorial computational processing cost in relation to how large the graphs are [1]. Notwithstanding the fact that only the compatible states shall allow the algorithm to yield a solution, such states are only a fragment of the state space S [1].

A consistency condition stating that the subgraphs corresponding to the partial mapping solution $R(i)$ are isomorphic graphs, i.e. $Q_s(i) \cong G_s(i)$, was constructed in regards to the graph and subgraph matching process [6]. It is essential to note that if a given subgraph $Q_s(i)$ at state i is non-isomorphic to another given subgraph $G_s(i)$ at state i, then this implies that the subgraphs $Q_s(i')$ and $G_s(i')$ at a newly generated state $i' = i \cup (v, u)$ will also be non-isomorphic since these are derived through adding vertices v and u to $Q_s(i)$ and $G_s(i)$, respectively [1].

Hence why it is important to centre the search procedures attention on finding only the consistent states to immensely reduce the computational processing cost of the algorithm. The VF2 algorithm searches through the state space S by implementing the concepts developed in the

depth-first search algorithm and a backtracking procedure [1].

For every repetition of the depth-first search procedure, the algorithm goes through the graphs to select a new possible pair of vertices (v, u) to construct successor states $i^{'}$ [1]. Note that the possible pair of vertices is not examined immediately after adding it to the subgraphs. The algorithm first verifies whether or not the new state $i^{'}$ is consistent, if it is indeed consistent then the pair is explored [1].

In [6], L.P. Cordella et al. formulated a set of consistency verification rules that are used to check the consistency of all the possible state space representation states i as it was initially detected that only a small subset of possible states is consistent with the desired matching procedure, in such a way that there are no conditions that prevent the chance of achieving a complete solution. Suppose the two given subgraphs $Q_s(i)$ and $G_s(i)$ satisfy the derived consistency verification rules of the desired matching procedure, then we can conclude that the state i shall be consistent. It is essential to mention that these verification rules are necessary, however they are not enough for the isomorphism check to hold. Regardless, the procedure assists the VF2 algorithm to significantly minimize the amount of states in the state space.

For example, if the algorithm is checking for a graph subgraph isomorphism detection, then the given intermediate state i is consistent if and only if the given subgraphs $Q_s(i)$ and $G_s(i)$ are isomorphic to each other. The VF2 algorithm introduces and implements these consistency verification rules such that only the consistent states are engineered and this shall be used to design new consistent states.

Furthermore, the initiation of a collection of rules that can verify beforehand if a given consistent state i contains inconsistent successor states can exceptionally minimize the amount of states modelled during the isomorphism procedure. L.P. Cordella et al. named these above mentioned verification rules the feasibility rules; in this section we shall first introduce the essential feasibility function denoted by $F(i, v, u)$ and then discuss the definition of the feasibility rules. A feasibility function $F(i, v, u)$ is satisfied if the addition of the pair of vertices (v, u) to the state space S holds for each and every feasibility rule [6].

Below is the universal expression of the feasibility function [6].

$$F(i, v, u) \;=\; F_{str}(i, v, u) \wedge F_{sem}(i, v, u). \tag{3.1}$$

As shown above in Equation (3.1), the feasibility function is made up of the structural feasibility function $F_{str}(i, v, u)$ which is dependent entirely on the adjacency and non-adjacency of vertices in the graphs; and the semantic feasibility function $F_{sem}(i, v, u)$ which is dependent on the vertex attributes. It is also employed to examine if the semantics of the given vertices is similar [6].

The structural feasibility function $F_{str}(i, v, u)$ examines the topological space of every given vertex by evaluating the consistency of three distinct subsets of vertices, that is to say:

- The neighboring vertices contained in the partial mapping solution $R(i)$.
- The neighboring vertices which are not contained in $R(i)$, however, are adjacent with

vertices contained in $R(i)$.

- The neighboring vertices which are neither contained nor connected in $R(i)$ [6].

We shall use two different sets of vertices, each for its respective graph. We use $V_{Q_s}(i) \subseteq V_Q(i)$ and $V_{G_s}(i) \subseteq V_G(i)$ to collect all the vertices that are already contained in $R(i)$, these sets shall be called the *core sets* [1]. Moreover, we use *end vertices* denoted by $\mathcal{T}_Q(i) \subseteq V_Q(i)$ and $\mathcal{T}_G(i) \subseteq V_G(i)$ whose vertices are adjacent to the vertices contained in the core sets. We shall now define and examine the feasibility rules which were formulated by L.P. Cordella et al. in [6] for consistency verification of the SSR state. It is essential to note that the idea of feasibility is one of the core concepts of the VF2 algorithm and its framework is entirely founded on it.

Let $\mathcal{C}_{pred}$ and $\mathcal{C}_{succ}$ denote the predecessor and successor feasibility rules, respectively; which are employed to examine the consistency of the partial mapping solution $R(i^{'})$ achieved after the insertion of the possible candidate pair (v, u) into the current partial mapping solution $R(i)$.

Similarly, let $\mathcal{C}_{in}$, $\mathcal{C}_{out}$ and $\mathcal{C}_{new}$ denote the feasibility rules implemented in reducing the search space by allowing them to execute an in advance analysis for inconsistent states, i.e., by first allowing the first two rules $\mathcal{C}_{in}$ and $\mathcal{C}_{out}$ to execute a 1-look-ahead rule in the searching procedure through which the VF2 algorithm distinguishes whether the new possible pair of vertices is feasible to construct a consistent state before analyzing it [1]; and then allowing the rule $\mathcal{C}_{new}$ to execute a 2-look-ahead rule [6].

These proposed feasibility rules were introduced on the basis that they might reduce computational processing costs by pruning states corresponding to partial matching solutions that do not satisfy the necessary graph morphism [7].

After introducing these five feasibility rules, the proposed syntactic feasibility function $F_{syn}(i, v, u)$ is now improved to satisfy these feasibility rules since this function is only dependent on the structure of the given graphs. Below is the improved version of the syntactic feasibility function [6],

$$F_{str}(i, v, u) = \mathcal{C}_{pred} \wedge \mathcal{C}_{succ} \wedge \mathcal{C}_{in} \wedge \mathcal{C}_{out} \wedge \mathcal{C}_{new}. \tag{3.2}$$

Let $\mathcal{T}_Q^{out}(i)$ and $\mathcal{T}_G^{out}(i)$ represent the vertex sets containing vertices that are not included in the partial mapping solution as yet, that are known to be the end vertices of edges beginning from $Q_s(i)$ and $G_s(i)$, respectively. Likewise, let $\mathcal{T}_Q^{in}(i)$ and $\mathcal{T}_G^{in}(i)$ represent the vertex sets containing vertices which are the source vertices of edges stopping into $Q_s(i)$ and $G_s(i)$, respectively; these vertices are not yet contained in the partial mapping. We now go through a detailed analysis of the feasibility rules, however we will need to use the sets below [6],

$$\mathcal{T}_Q(i) = \mathcal{T}_Q^{in}(i) \bigcup \mathcal{T}_Q^{out}(i); \tag{3.3}$$

$$\mathcal{V}_Q(i) = V_Q - V_{Q_s}(i) - \mathcal{T}_Q(i). \tag{3.4}$$

Likewise, the expressions in equation (3.3) and equation (3.4) must be satisfied for the sets $\mathcal{T}_G$ and $\mathcal{V}_G$. These sets are used in deriving the feasibility rules,

$$\mathcal{T}_G(i) = \mathcal{T}_G^{in}(i) \bigcup \mathcal{T}_G^{out}(i); \tag{3.5}$$

$$\mathcal{V}_G(i) = V_G - V_{G_s}(i) - \mathcal{T}_G(i). \tag{3.6}$$

Below, we shall derive the feasibility rules just as in [6] for the subgraph isomorphism problem. The predecessor and successor feasibility rules $\mathcal{C}_{pred}$ and $\mathcal{C}_{succ}$ which are used for consistency of the partial mapping solution are defined below [6].

The *predecessor feasibility rule* denoted by $\mathcal{C}_{pred}(i, v, u)$ is satisfied if and only if for each vertex v_1 contained in the vertex set $V_{Q_s}(i) \cap \mathcal{M}(Q, v)$, there exists a predecessor vertex u_1 contained in the predecessor set $\mathcal{M}(G, u)$ such that there is a relation (v_1, u_1) contained in the partial mapping solution $R(i) \subseteq R \subseteq V_Q \times V_G$. Similarly, for each vertex u_1 contained in the vertex set $V_{G_s}(i) \cap \mathcal{M}(G, u)$, there exists a predecessor vertex v_1 contained in the predecessor set $\mathcal{M}(Q, v)$ such that there is a relation (v_1, u_1) contained in the partial mapping solution $R(i)$ [6].

Furthermore, the *successor feasibility rule* represented by $\mathcal{C}_{succ}(i, v, u)$ holds if and only if for each vertex v_1 contained in the vertex set $V_{Q_s}(i) \cap \mathcal{S}(Q, v)$, there exists a successor vertex u_1 contained in the successor set $\mathcal{S}(G, u)$ such that there is a relation (v_1, u_1) contained in $R(i)$; and for each vertex u_1 contained in the vertex set $V_{G_s}(i) \cap \mathcal{S}(G, u)$, there exists a successor vertex v_1 contained in the successor set $\mathcal{S}(Q, v)$ such that there is a relation (v_1, u_1) contained in the partial mapping solution $R(i)$ [6].

The two feasibility rules mentioned above are implemented to test for consistency of a given state corresponding to the partial mapping $R(i)$ which is associated with the two given subgraphs $Q_s(i)$ and $G_s(i)$.

We further define the feasibility rules which are denoted by $\mathcal{C}_{in}$, $\mathcal{C}_{out}$ and $\mathcal{C}_{new}$ in [6]. These three feasibility rules are used for pruning a given search tree; that is to say, $\mathcal{C}_{in}$, $\mathcal{C}_{out}$ and $\mathcal{C}_{new}$ implement an in advance check in the search process [6]. Supposing these rules are not satisfied then it is definite that the algorithm will not find a target state while it is searching for a successor state of i. Below are conditions for the feasibility rules to be satisfied [6]:

The feasibility rule denoted by $\mathcal{C}_{in}(i, v, u)$ is satisfied if and only if $|\mathcal{S}(Q, v) \cap \mathcal{T}_Q^{in}(i)| \geq |\mathcal{S}(G, u) \cap \mathcal{T}_G^{in}(i)|$, and $|\mathcal{M}(Q, v) \cap \mathcal{T}_Q^{in}(i)| \geq |\mathcal{M}(G, u) \cap \mathcal{T}_G^{in}(i)|$.

Moreover, the feasibility rule $\mathcal{C}_{out}(i, v, u)$ holds if and only if $|\mathcal{S}(Q, v) \cap \mathcal{T}_Q^{out}(i)| \geq |\mathcal{S}(G, u) \cap \mathcal{T}_G^{out}(i)|$ and $|\mathcal{M}(Q, v) \cap \mathcal{T}_Q^{out}| \geq |\mathcal{M}(G, u) \cap \mathcal{T}_G^{out}(i)|$.

Lastly, the following feasibility rule denoted by $\mathcal{C}_{new}(i, v, u)$ holds if and only if $|\mathcal{V}_Q(i) \cap \mathcal{M}(Q, v)| \geq |\mathcal{V}_G(i) \cap \mathcal{M}(G, u)|$ and $|\mathcal{V}_Q(i) \cap \mathcal{S}(Q, v)| \geq |\mathcal{V}_G(i) \cap \mathcal{S}(G, u)|$.

It is significant to note that if we consider the case for graph isomorphism rather than the subgraph isomorphism, the above given expressions for the last three feasibility rules denoted by $\mathcal{C}_{in}(i, v, u)$, $\mathcal{C}_{out}(i, v, u)$, and $\mathcal{C}_{new}(i, v, u)$ changes its mathematical operator $\geq$ to the mathematical operator $=$; this is to consider graphs of the same order and size. However, the rules given by $\mathcal{C}_{pred}(i, v, u)$ and $\mathcal{C}_{succ}(i, v, u)$ remain the same.

We shall now discuss how the set of all distinct possible candidate vertex pair $\rho = (v, u)$ is

constructed, and how these new candidate vertex pairs are added to the current partial mapping $R(i)$. The VF2 algorithm determines two candidate pair sets denoted by $\mathcal{P}_Q(i)$ and $\mathcal{P}_G(i)$, where the algorithm will look for new candidate vertex pairs and thus test whether they are feasible by applying the feasibility rules.

Below, we define the two candidate pair sets [1] :

$$\mathcal{P}_Q(i) = \begin{cases} \mathcal{T}_Q(i), & if\ \mathcal{T}_Q(i) \neq \emptyset \wedge \mathcal{T}_G(i) \neq \emptyset \\ \mathcal{V}_Q(i), & otherwise \end{cases} \tag{3.7}$$

$$\mathcal{P}_G(i) = \begin{cases} \mathcal{T}_G(i), & if\ \mathcal{T}_Q(i) \neq \emptyset \wedge \mathcal{T}_G(i) \neq \emptyset \\ \mathcal{V}_G(i), & otherwise \end{cases} \tag{3.8}$$

If we consider the case for directed graphs, the above candidate pair sets were improved to get the sets in the equations below, just as in [1] and [6] :

$$\mathcal{P}_Q(i) = \begin{cases} \mathcal{T}_Q^{out}(i), & if\ \mathcal{T}_Q^{out}(i) \neq \emptyset \wedge \mathcal{T}_G^{out}(i) \neq \emptyset \\ \mathcal{T}_Q^{in}(i), & if\ \mathcal{T}_Q^{in}(i) \neq \emptyset \wedge \mathcal{T}_G^{in}(i) \neq \emptyset \\ \mathcal{V}_Q(i), & otherwise \end{cases} \tag{3.9}$$

$$\mathcal{P}_G(i) = \begin{cases} \mathcal{T}_G^{out}(i), & if\ \mathcal{T}_Q^{out}(i) \neq \emptyset \wedge \mathcal{T}_G^{out}(i) \neq \emptyset \\ \mathcal{T}_G^{in}(i), & if\ \mathcal{T}_Q^{in}(i) \neq \emptyset \wedge \mathcal{T}_G^{in}(i) \neq \emptyset \\ \mathcal{V}_G(i), & otherwise \end{cases} \tag{3.10}$$

As shown by the above equations in Equation (3.9) and Equation (3.10), the candidate pair sets $\mathcal{P}_Q(i)$ and $\mathcal{P}_G(i)$ are derived by considering every vertex pair (v, u), where vertex v is an element of $\mathcal{T}_Q^{out}(i)$ and the other vertex u is an element of $\mathcal{T}_G^{out}(i)$, except for the case when these sets are empty sets [6]. Similarly, these sets can be achieved by taking into consideration the sets $\mathcal{T}_Q^{in}(i)$ and $\mathcal{T}_G^{in}(i)$.

Below is an advanced description of the original VF2 algorithm developed in [6].

Listing 3.1: **The VF2 isomorphism algorithm**

```
Procedure Match(i)
    Input : an intermediate state i; initializing state i_{0} with an empty partial mapping R(i_{0})
    Output : the mappings between the two given graphs

    if R(i) explores all the vertices of the graph G then
        Output R(i)
    Else
        Compute the candidate pair sets P_{Q}(i), P_{G}(i) to be added in the partial mapping solution R(i);
            for every candidate vertex pair (v,u) in P_{Q}(i), P_{G}(i);
                if the feasibility rules are satisfied for the addition of candidate vertex pairs (v,u) in
                    R(i);
                Then compute the new state i' = i union (v,u) achieved by adding (v,u) to the partial
                    mapping R(i);
                Call Match(i)
            End if
        End for
        Restore data structures
    End if
End Procedure Match
```

In what follows, a detailed description of the matching strategy implemented in the VF2 algorithm for graph and subgraph isomorphism. Here, note that the inputs given are an initial state i, and two distinct graphs $Q = (V_Q, E_Q)$ and $G = (V_G, V_G)$. This procedure is a based on a boolean strategy; that is to say, it yields true if "there is one and only one" matching solution, however it returns false if the matching solution does not exist.

Listing 3.2: **The structure of the isomorphism strategy implemented by the VF2 algorithm [1]**

```
1: function Match(i, Q, G)
2:
3: if IsTarget(i) then
4:     return True
5: end if
6:
7: if IsTerminate(i) then
8:     return False
9: end if
10:
11: Set v = k and u = k
12: (v',u') = GetNextCandidate(i,(v,u),Q,G)
13: while v' is not equal k, and u' is not              equal k do;
14:             if IsFeasible((v',u'),Q,G) then
15:                 i' = i union (v',u')
16:                 if Match(i',Q,G) is True then
17:                     return True
18:                 end if
19:             end if
20:                 (v',u') = GetNextCandidate(i,(v',u'),Q,G)
21:             end while
22:       return False
23: end function
```

The VF2 algorithm implements a relation/a mapping which performs a correspondence of two distinct vertices, each vertex from two distinct graphs. The procedure in the VF2 algorithm looks for new candidate vertex pairs, however, the algorithm needs to consider pairs that already exist and thus is required to disregard any candidate vertex pairs contained in the candidate pair sets.

The VF2 algorithm also employs the depth-first search algorithm to traverse through the graphs. The depth-first search algorithm assists the VF2 algorithm in working with vertex cycles and vertex paths that may be repetitive in the search procedure. This implies that, while searching through the state space S, the algorithm can prevent itself from reconstructing already visited states.

Chapter 4

4.1 Simulation Analysis

The motivation for this section lies in examining all the possible factors influencing the computational processing time of the Ullmann subgraph isomorphism algorithm; in turn determining the algorithm's efficiency. A few of the factors that influence the cost-effectiveness of the algorithm are considered below.

- The order of the pattern graph, that is $|V_Q|$.
- The order of the data graph $|V_G|$.
- The degree of vertices in the data graph G.
- The degree of vertices in the pattern graph Q.

In this section, we explore the above mentioned factors to study the algorithm's behaviour for a given set of pattern graphs $Q = (V_Q, E_Q)$. Here, we consider one large data graph $G = (V_G, E_G)$ as a reference to search in; along with its adjacency matrix B_G. We also consider 15 distinct Erdös Rényi random pattern graphs to search for subgraph isomorphism detection in the large data graph. The 15 graphs differ in the number of vertices and the adjacency of vertices; these graphs are available on request.

In our implementation we use adjacency matrices to examine subgraph isomorphism detection, rather than employing an adjacency list respresentation. We shall only go through the implementation of the Ullmann's algorithm for subgraph isomorphism detection; we wrote this implementation in Java as our own original work, however, we used the algorithm's original framework developed by Ullmann in [19], and the improvements of the Ullmann's algorithm derived in [3].

The random graphs we considered were constructed in python by using the NetworkX random graph generator, and we further construct their respective adjacency matrices. Each graph has its respective adjacency matrix and in turn a respective initialized candidate permutation matrix.

Below is the large adjacency matrix of the data graph $G = (V_G, E_G)$ which we considered for isomorphism detection by iteratively searching through B_G, and at the same time satisfying the condition specified in Equation (2.24).

$$
\mathrm{B}_G = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

The above adjacency matrix B_G is rearranged by using the condition in equation (2.10) in order to achieve other possible candidate adjacency matrices whose dimensions are the same as the adjacency matrix of $Q = (V_Q, E_Q)$.

We consider 15 random pattern graphs $Q = (V_Q, E_Q)$ which contain a distinct number of vertices in V_Q. We begin the procedure by considering a pattern graph containing two vertices for the subgraph isomorphism detection and in turn examining the run-time of the procedure. All the other graphs are available on request, along with the implementation described in Appendix A and B.

According to Ullmann in [19], the run-time complexity of his algorithm resulted into an $\mathcal{O}(|V_Q|^2)$.

Nontheless, our procedure results into an $\mathcal{O}(|V_Q|)$ execution run-time complexity, however, our implementation detects not only the subgraph isomorphism (exact matching candidates), but also the inexact matching candidate (partial matching candidates).

In [19], Ullmann constructed a graphical representation which specifies a parabolic graphical behaviour. In this dissertation, we intend to compare our results with those described by Ullmann in [19]. It is of great importance to note that our procedure employs Ullmann's framework, however, we programmed our procedure in such a way that the algorithm searches for all matches; be it exact matches or in-exact matches, and thus in turn verifies whether the matches are true or not by the use of a Boolean statement.

We execute our procedure on all 15 distinct random pattern graphs in order to determine the run-time complexity of our implementation for every given pattern graph $Q = (V_Q, E_Q)$. Table 4.1 shows the number of vertices contained in their respective vertex sets of 15 distinct graphs; starting from small to large, along with the time taken for the algorithm to simultaneously process the adjacency matrices of the given pattern graphs $Q = (V_Q, E_Q)$ and the large data graph $G = (V_G, E_G)$ so that candidate adjacency matrices can be constructed and in turn detecting matches between two given graphs.

Table 4.1: Empirical results

$\lvert V_Q \rvert$	**Time Taken**	**Number of matches**	**Number of exact matches**	**Partial Matches**
2	0 minutes 0.25 seconds	870 matches	342	428
3	0 minutes 36 seconds	19656 matches	653	19002
4	0 minutes 55 seconds	212520 matches	112500	100020
5	2 minutes 56 seconds	1395360 matches	60	1395300
6	3 minutes 58 seconds	1396555 matches	2	1396553
7	4 minutes 02 seconds	1397050 matches	0	1397050
8	4 minutes 27 seconds	1398145 matches	0	1398145
9	5 minutes 12 seconds	1399754 matches	0	1399754
10	6 minutes 45 seconds	1421240 matches	0	1421240
15	NAN	NAN		
20	NAN	NAN		
21	NAN	NAN		
24	NAN	NAN		
28	NAN	NAN		
30	NAN	NAN		

In the case of the pattern graphs of order 15, 20, 21, 24, 28 and 30, the source code executes and at some point in time it crashes due to memory allocation to store all the generated candidate permutation matrices along with their corresponding candidate adjacency matrices, which are in turn compared with the adjacency matrices of the respective pattern graphs.

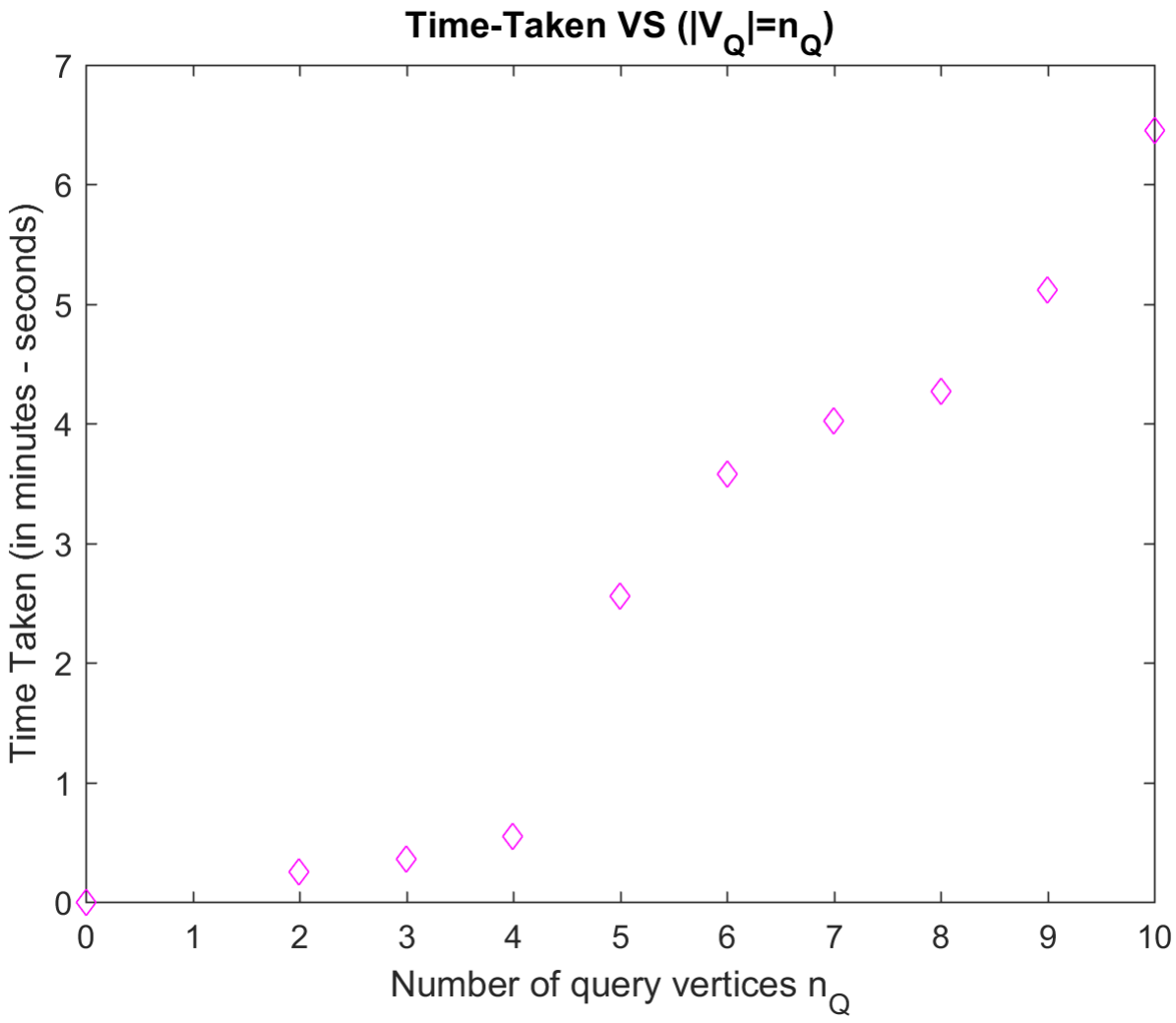


Figure 4.1: Time-Taken vs $|V_Q|$

It is essential to note that our re-formulation of the Ullmann algorithm allows for both exact matching and partial matching detection, i.e., inexact matching. Whereas the original algorithm searches for strict exact matching between two given graphs. Our implementation yields the total number of matches and then divides it into the total number of exact matches and the total number of partial matches.

In Figure 4.1, we have a plot of the experimental data which shows how the execution runtime of the Ullmann algorithm increases. In Figure 4.2, we plot a line of best fit which predicts the efficiency of the algorithm. However, we can not conclude much using such a small dataset.

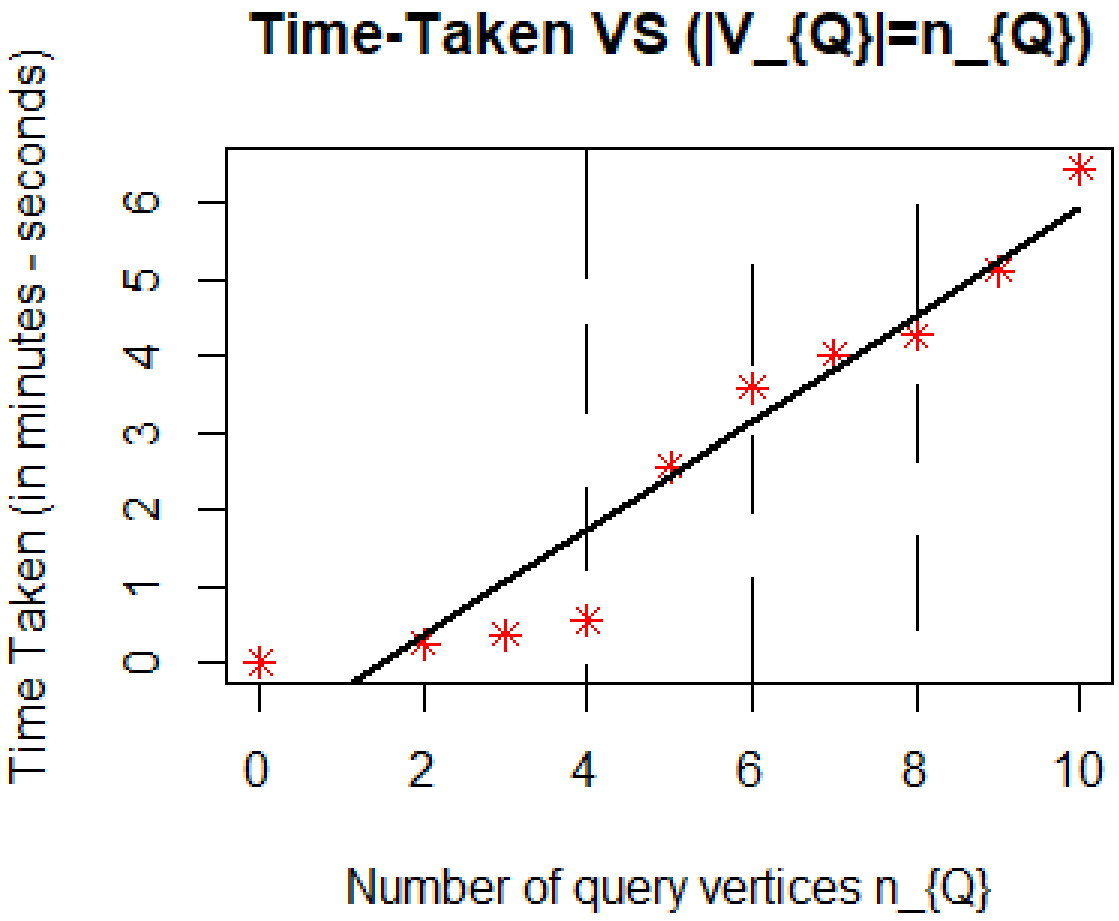


Figure 4.2: Efficiency Prediction Using Regression Line.

Chapter 5

5.1 Conclusion

The subgraph isomorphism detection problem has been an essential topic in the field of Computer Science and Theoretical Graph Theory. The purpose of this dissertation is to extensively examine algorithms developed for the graph and subgraph isomorphism detection problems; in particular, in this dissertation we study the algorithm proposed by J. R. Ullmann in 1976. According to [19], the Ullmann algorithm works perfectly for randomly generated graphs since it detects all isomorphisms within $|V_Q|$ execution time-complexity. This in turn satisfies the criterion proposed by Corniel and Gotliebs in [9], which states that an algorithm is efficient provided that its execution time is close to the power of $|V_Q|$, that is to say $|V_Q|^n$ where $n \in \mathbb{Z}$.

In this dissertation, we tried to re-formulate the Ullmanns algorithm for the graph and subgraph isomorphism detection. However, some of the changes we made to the algorithm reduced the algorithms efficiency than that proposed in [9]. The one advantage we discovered about the Ullmann algorithm is that it can cope with randomly generated undirected graphs for the graph and subgraph isomorphism detection. However, the algorithm becomes very slow as the number of vertices and edges in the pattern graph $Q = (V_Q, E_Q)$ are increased. Note that the order and size of the data graph does not change, however, we search for pattern graphs of different orders and sizes.

In Table 4.1, our empirical results show that the time taken for the algorithm to run increases with an increase in the number of vertices in V_Q. However, based on the changes we implemented in the source code and due to hardware, the algorithm fails to exit; that is to say, the algorithm crashes for pattern graphs of order greater or equal to 11. Hence this implies that our implementation is limited to pattern graphs of order $|V_Q| \leq 10$.

According to [6], the VF2 algorithm is a much faster algorithm and works perfectly with directed graphs, however, it is not as efficient for randomly generated undirected graphs. In this dissertation we extensively employed the Ullmann's algorithm to study its efficiency; but we were unable to employ the VF2 algorithm due to the lack of experimental data and lack of computational analysis techniques.

In future research, we intend to revisit the graph and subgraph isomorphism detection problems, and we hope to improve the execution run-time of the existing isomorphism detection

algorithms; and perhaps formulate our own framework for graph and subgraph isomorphism detection. We also hope to show that the efficiency curve drops as we increase the order and size of the pattern graphs with much larger data structures and datasets. We intend to look into several distinct ways to improve the execution run-time of the Ullmann algorithm by executing the Ullmann algorithm as a parallel computing isomorphism detection problem.

Here, we would partition the problem into sub-problems that can be simultaneously solved to reduce the execution run-time of the algorithm; in turn improving the algorithms efficiency.

Bibliography

[1] Vincenzo Carletti. "Exact and Inexact Methods for Graph Similarity in Structural Pattern Recognition PhD thesis of Vincenzo Carletti." PhD thesis. 2016.

[2] Gary Chartrand, Linda Lesniak, and Ping Zhang. *Graphs & digraphs*. Chapman and Hall/CRC, 2010.

[3] Uroš Čibej and Jurij Mihelič. "Improvements to ullmann's algorithm for the subgraph isomorphism problem". In: *International Journal of Pattern Recognition and Artificial Intelligence* 29.07 (2015), p. 1550025.

[4] Uroš Čibej and Jurij Mihelič. "Search strategies for subgraph isomorphism algorithms". In: *International Conference on Applied Algorithms*. Springer. 2014, pp. 77–88.

[5] Donatello Conte et al. "Thirty years of graph matching in pattern recognition". In: *International journal of pattern recognition and artificial intelligence* 18.03 (2004), pp. 265–298.

[6] Luigi P Cordella et al. "A (sub) graph isomorphism algorithm for matching large graphs". In: *IEEE transactions on pattern analysis and machine intelligence* 26.10 (2004), pp. 1367–1372.

[7] Luigi P Cordella et al. "Graph matching: a fast algorithm and its evaluation". In: *Proceedings. Fourteenth International Conference on Pattern Recognition (Cat. No. 98EX170)*. Vol. 2. IEEE. 1998, pp. 1582–1584.

[8] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[9] D. G. Corneil and C. C. Gotlieb. "An Efficient Algorithm for Graph Isomorphism". In: *J. ACM* 17.1 (Jan. 1970), pp. 51–64. ISSN: 0004-5411. DOI: 10.1145/321556.321562. URL: http://doi.acm.org/10.1145/321556.321562.

[10] Lixin Fu and Surya Prakash R Kommireddy. "Labeled Subgraph Matching Using Degree Filtering". In: *Proceedings of the International Conference on Data Mining (DMIN)*. The Steering Committee of The World Congress in Computer Science, Computer â. 2013, p. 1.

[11] Merrick Furst, John Hopcroft, and Eugene Luks. *Polynomial-Time Algorithms for Permutation Groups.* Tech. rep. CORNELL UNIV ITHACA NY DEPT OF COMPUTER SCIENCE, 1980.

[12] Alan Gibbons. *Algorithmic graph theory*. Cambridge university press, 1985.

[13] Jonathan L Gross and Jay Yellen. *Graph theory and its applications*. Chapman and Hall/CRC, 2005.

[14] Alpár Jüttner. "Algorithms for Matching Biological Graphs". In: (2016).

[15] Ulrich Knauer. *Algebraic graph theory: morphisms, monoids and matrices*. Vol. 41. Walter de Gruyter, 2011.

[16] Jinsoo Lee et al. "An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases". In: *Proc. VLDB Endow.* 6.2 (Dec. 2012), pp. 133–144. ISSN: 2150-8097. DOI: 10.14778/2535568.2448946. URL: http://dx.doi.org/10.14778/2535568.2448946.

[17] Bruno T Messmer and Horst Bunke. *Subgraph isomorphism in polynomial time*. Universität Bern. Institut für Informatik und Angewandte Mathematik, 1995.

[18] Joshua Mitchell et al. "Development and In silico Evaluation of Large-Scale Metabolite Identification Methods using Functional Group Detection for Metabolomics". In: *Frontiers in Genetics* 5 (July 2014). DOI: 10.3389/fgene.2014.00237.

[19] Julian R. Ullmann. "An Algorithm for Subgraph Isomorphism". In: *J. ACM* 23.1 (1976), pp. 31–42. DOI: 10.1145/321921.321925. URL: http://doi.acm.org/10.1145/321921.321925.

[20] Julian R. Ullmann. "Bit-vector Algorithms for Binary Constraint Satisfaction and Subgraph Isomorphism". In: *J. Exp. Algorithmics* 15 (Feb. 2011), 1.6:1.1–1.6:1.64. ISSN: 1084-6654. DOI: 10.1145/1671970.1921702. URL: http://doi.acm.org/10.1145/1671970.1921702.

[21] Gabriel Valiente. *Algorithms on trees and graphs*. Berlin; New York: Springer, 2002. ISBN: 3540435506 9783540435501 3642078095 9783642078095. URL: http://www.amazon.de/Algorithms-Trees-Graphs-Gabriel-Valiente/dp/3540435506.

[22] Eric W Weisstein. *Permutation Matrix*. 2019-05-14. URL: http://mathworld.wolfram.com/PermutationMatrix.html.

Chapter 6

6.1 Appendix A

For Example 1

Listing 6.1: **The main Java code for** 5×11 **candidate permutation matrix**

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file , choose Tools | Templates
 * and open the template in the editor.
 */
package Final_Ullmans;

import java.util.ArrayList;

/**
 *
 * @author Kellen
 */
public class Main
{
    /**
     *
     * This method is used to copy an array list of permutations into a larger list
     * that allow us to store all the different permutations for a certain level
     *
     * @param A - this is the big array list
     * @param B - this is the list to be copied to the bigger list
     * @return the bigger list 'A'
     */
    public static ArrayList<Perm> BigList (ArrayList<Perm> A , ArrayList<Perm> B)
    {
        for (int i = 0; i < B.size(); i++) {
            A.add(B.get(i));

        }
        return A;
    }

    public static void main(String[] args)
```

```
    {

        int P0[][]=    {{1,1,0,0,0,1,1,0,0,0,1},
                        {1,1,1,1,0,1,1,1,1,1,1},
                        {1,1,1,1,0,1,1,1,1,1,1},
                        {1,1,1,1,0,1,1,1,1,1,1},
                        {1,1,1,1,1,1,1,1,1,1,1}};

       ArrayList<Perm> TempList = new ArrayList<Perm>();
       ArrayList<Perm> TempList2 = new ArrayList<Perm>();


       int pr = 5;
       int pc = 11;

       Perm p = new Perm (pr,pc);


       p.setOriginal(P0);


       System.out.println("");



       PermArraylist list = new PermArraylist();
       System.out.println("The initialised candidate permutation matrix");
       p.printOriginal();
       System.out.println("");



       System.out.println("Level is Starting");
       TempList = list.getPermsByLevel(pr, pc,0, P0);
        list = new PermArraylist(TempList);

       for(int j = 1;j < pr ;j++)
       {
        System.out.println("************************* this is level "+(j) +"**********************");
        TempList2 = new ArrayList<Perm>();
        for (int i = 0; i <list.getPermList().size(); i++)
           {

               TempList = list.getPermsByLevel(pr, pc,j, list.getPermList().get(i).getOriginal());
               TempList2 = BigList(TempList2,TempList);

           }
        System.out.println("size is"+TempList2.size());
         list = new PermArraylist(TempList2);


       }

    }

}
```

Listing 6.2: **The construction of a permutation class for all candidate permutation matrices**

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Final_Ullmans;

/**
 *
 * @author Kellen
 */

public class Perm
{
    private int r,c;
    private int perm[][] = new int[r][c];

    private int original[][] = new int[r][c];

    private int level;

    /**
     *
     * @param r this is the row size of the matrix
     * @param c this is the column size of the matrix
     */
    public Perm(int r, int c)
    {
        this.r = r;
        this.c = c;

        perm = new int[r][c];
        original = new int[r][c];
    }



    /**
     * This is used to print the original Matrix
     */
    public void printOriginal()
    {
        for (int i = 0; i < r; i++)
        {
            for (int j = 0; j < c; j++)
            {
                System.out.print(original[i][j]+" ");

            }
            System.out.println("");

        }

    }
```

```
/**
 * This is used to print the permutation Matrix
 */
 public  void printPermutation()
{
    for (int i = 0; i < r; i++)
    {
        for (int j = 0; j < c; j++)
        {
            System.out.print(perm[i][j]+" ");

        }
        System.out.println("");

    }

}

 /**
  * This method is used to make a deep copy of a matrix
  *
  * @param matrix this is the matrix going to be copied
  * @param r this is the row size
  * @param c this is the column size
  * @return this return a deep copy
  */
 public int [][] deepCopy (int matrix[][],int r,int c)
{
    int M [][] = new int [r][c];
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            M[i][j] = matrix[i][j];

        }

    }
    return M;
}

 /**
  *
  * @param idcol
  * @param col
  * @param pr
  * @return
  */
 public static boolean IsIdCol(int[] idcol,int col,int pr)

{
    boolean flag = false;

    for (int i = 0; i < pr; i++)
    {
        if(idcol[i]==col)
        {
            flag = true;
```

```
            }

        }
        return flag;
    }
     /**
      * this is used to create a permutation matrix from the original matrix
      *
      *
      * @param row - this is size of the rows in the permutation
      * @param col - this is the size of the columns in the permutations
      * @return - this will return a permutation matrix
      */

       public int [][] permutationz(int row ,int col)
    {
        perm = this.getOriginal();
        for (int i = 0; i < c; i++)
        {
            if (i==col)
            {
                perm[row][i]=1;
            }else
            {
               perm[row][i]=0;
            }

        }
        return perm;
    }

    /**
     *
     * @return r - row size
     */
    public int getR() {
        return r;
    }
    /**
     *
     * @param r - set the row size
     */
    public void setR(int r) {
        this.r = r;
    }
    /**
     *
     * @return - return the column size
     */
    public int getC() {
        return c;
    }
    /**
     *
     * @param c - set the number of columns
     */
```

```
    public void setC(int c) {
        this.c = c;
    }
    /**
     *
     * @return a permutation matrix
     */
    public int [][] getPerm() {
        return perm;
    }
    /**
     *
     * @param perm - this will set the permutation matrix of the class
     */
    public void setPerm(int[][] perm) {
        this.perm = perm;
    }
    /**
     *
     * @return a the original matrix before permutation has taken place
     */
    public int [][] getOriginal() {
        return original;
    }
    /**
     *
     * @param original set the orginal matrix
     */
    public void setOriginal(int [][] original) {
        this.original = original;
    }
    /**
     *
     * @return the level being computed
     */
    public int getLevel() {
        return level;
    }
    /**
     *
     * @param level set the level which is being computed
     */
    public void setLevel(int level) {
        this.level = level;
    }

}
```

Listing 6.3: **Construction of a set of all candidate permutation matrices**

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Final_Ullmans;

import java.util.ArrayList;

/**
 *
 * @author Kellen
 */
public class PermArraylist
{
    private ArrayList<Perm> permList = new ArrayList<Perm>();
    private ArrayList<Perm> TempList = new ArrayList<Perm>();
    private int B[][] = {   {0,1,1,0,0,0,0,0,0,0,1},
                            {1,0,0,1,0,0,0,0,0,1,1},
                            {1,0,0,0,0,1,0,0,0,0,0},
                            {0,1,0,0,0,0,1,0,0,0,0},
                            {0,0,0,0,0,1,0,0,0,0,0},
                            {0,0,1,0,1,0,0,1,1,0,0},
                            {0,0,0,1,0,0,0,1,1,1,1},
                            {0,0,0,0,0,1,1,0,0,0,0},
                            {0,0,0,0,0,1,1,0,0,0,0},
                            {0,1,0,0,0,0,1,0,0,0,0},
                            {1,1,0,0,0,0,1,0,0,0,0}};

    private int Q [][]={  {0,1,1,1,0},
                          {1,0,0,1,0},
                          {1,0,0,0,1},
                          {1,1,0,0,0},
                          {0,0,1,0,0}};



    public PermArraylist(ArrayList pL)
    {
        this.permList = pL;
    }

    public PermArraylist()
    {

    }

   public boolean IsIdCol(int[] idcol,int col,int pr)

    {
        boolean flag = false;

        for (int i = 0; i < pr; i++)
```

```
        {
            if (idcol[i]==col)
            {
                flag = true;
            }

        }
        return flag;
    }

    public int idCol(int matrix [][], int row,int col)
    {
        for (int i = 0; i <col ; i++)
        {
            if (matrix[row][i]==1)
            {
                return i;
            }


        }
       return -1;
    }

    public static boolean idRow (int matrix [][], int row,int col)
    {
        boolean flag = false ;
        int count = 0;
        for (int i = 0; i <col; i++)
        {
            int val = matrix[row][i];

            if (val == 1)
            {
                count++;
            }

        }

        if (count<=1)
        {
            flag = true;
        }

        return flag;
    }

    public  ArrayList<Perm> getPermsByLevel(int r ,int c,int lvl,int [][] ori)
    {
        this .TempList.clear();
        boolean idRows[] = new boolean[r];
        int idCols[] = new int [r];

        Perm p = new Perm (r,c);

        int [][] permuatation = new int [r][c];
```

```
p.setOriginal(ori);

p.setLevel(lvl);
Perm temp = new Perm (r,c);


for (int i = 0; i < r; i++)
{
    idRows[i]= idRow(ori,i,c);
}

for (int i = 0; i < r; i++)
{
    if (idRows[i]==true)
    {
        idCols[i]=idCol(ori,i,c);
    }else
    {
        idCols[i]=-1;
    }

}

for (int i = 0; i < idRows.length; i++) // this checks the rows with the condition of having one , '1'
    in each row
{
    System.out.println( "Is this row a ID row " + i + " : " + idRows[i]);
}

for (int i = 0; i < idCols.length; i++) // this checks which column has at most '1' in it
{
    System.out.println("The row "+i+ " column : "+idCols[i]);


}

for (int i = 0; i < c ; i++)
{
    temp = new Perm (r,c);
    temp.setOriginal(temp.deepCopy(ori, r, c));

    if (temp.getOriginal()[lvl][i]==1)
    {


        if (IsIdCol(idCols,i,r)==false)
        {


            temp.setOriginal(temp.deepCopy(ori, r, c));
            permuatation = temp.permutationz(lvl,i);
            if (lvl ==r-1)
            {
                System.out.println("This is the candidate permutation matrix");
            }
            temp.printPermutation();
```

```
if (lvl ==r-1)
{
    int A [][] = permuatation;
    MatrixMulti MM = new MatrixMulti ();

    int AB [][] = MM.multipli(A, B);

    int AT [][] = MM.transposeMatrix(A);

    int Mat [][] = MM.multipli(AB, AT);

     System.out.println("This is the resulting candidate ajacency matrix");
     for (int j = 0; j < Mat.length; j++)
     {
         for (int k = 0; k <Mat[0].length; k++)
         {
             System.out.print(Mat[j][k]+" ");
         }
         System.out.println("");
     }

     System.out.println("END");

     boolean C [][]= new boolean [Mat.length][Mat[0].length];
     int count =0;

     for (int k = 0; k < Mat.length; k++)
     {
        for (int j = 0; j < Mat[0].length; j++)
        {
            if (Mat[k][j]==Q[k][j])
            {
                C[k][j] = true;
                count++;
            } else
            {
                C[k][j] = false;
            }


        }

    }

        if (count == (r*c))
        {
            System.out.println("Exact Match");
        } else if (count > 0)
        {
            System.out.println("Partial Vertex Similarities");
        } else {
            System.out.println("Not Even Close");
        }


}
```

```
                    Perm temp2 = temp;
                    temp2.setPerm(temp.deepCopy(temp.permutationz(lvl,i),r,c));
                    permList.add(temp2);
                    TempList.add(temp2);
                    System.out.println("");

                }

            }


        }
        return TempList;

    }

    public Perm getPerm (int i)
    {
        return permList.get(i);
    }

    public  ArrayList<Perm> getPermList() {
        return permList;
    }

    public  ArrayList<Perm> getTempList() {
        return TempList;
    }

    public  void setPermList(ArrayList<Perm> permList) {
        this.permList = permList;
    }

    public  void setTempList(ArrayList<Perm> permList) {
        this.TempList = permList;
    }

}
```

Listing 6.4: **Implementing the condition in equation (10) to get candidate adjacency matrices**

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Final_Ullmans;

/**
 *
 * @author Kellen
 */
public class MatrixMulti
{

    public MatrixMulti() {
    }

    /**
     * This method is used to multiply two matrices together n return their product
     *
     * @param A - this is the first matrix to be multiplied
     * @param B - this is the second matrix to be multiplied
     * @return - this will return matrix multiplication between A and B = AB
     */
    public int [][] multipli(int [][] A, int [][] B) {

        int aRows = A.length;
        int aColumns = A[0].length;
        int bRows = B.length;
        int bColumns = B[0].length;

        if (aColumns != bRows) {
            throw new IllegalArgumentException("A:Rows: " + aColumns + " did not match B:Columns " +
                    bRows + ".");
        }

        int [][] C = new int[aRows][bColumns];
        for (int i = 0; i < aRows; i++) {
            for (int j = 0; j < bColumns; j++) {
                C[i][j] = 0;
            }
        }

        for (int i = 0; i < aRows; i++) { // aRow
            for (int j = 0; j < bColumns; j++) { // bColumn
                for (int k = 0; k < aColumns; k++) { // aColumn
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }

        return C;
    }
    /**
```

```
     * This method is used to take a matrix n return a transposed version of that matrix
     *
     * @param m this is the matrix that needs to be transposed
     * @return return a transposed version of matrix 'm'
     */
    public int [][] transposeMatrix(int [][] m){
        int [][] temp = new int[m[0].length][m.length];
        for (int i = 0; i < m.length; i++)
            for (int j = 0; j < m[0].length; j++)
                temp[j][i] = m[i][j];
        return temp;
    }

}
```

6.2 Appendix B

For Example 2

Listing 6.5: **The main Java code for** 3×4 **candidate permutation matrix**

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file , choose Tools | Templates
 * and open the template in the editor.
 */
package Final_Ullmans;

import java.util.ArrayList;

/**
 *
 * @author Kellen
 */
public class Main
{
    /**
     *
     * This method is used to copy an array list of permutations into a larger list
     * that allow us to store all the different permutations for a certain level
     *
     * @param A – this is the big array list
     * @param B – this is the list to be copied to the bigger list
     * @return the bigger list 'A'
     */
    public static ArrayList<Perm> BigList (ArrayList<Perm> A , ArrayList<Perm> B)
    {
        for (int i = 0; i < B.size(); i++) {
            A.add(B.get(i));

        }
        return A;
    }

    public static void main(String[] args)
    {
        //NB : Example2 derived in
            paper----------------------------------------
         int P0[][] = { {1,0,1,1},
                        {1,1,1,1},
                        {0,1,0,0}};


        ArrayList<Perm> TempList = new ArrayList<Perm>();
        ArrayList<Perm> TempList2 = new ArrayList<Perm>();


        int pr = 3;
        int pc = 4;
```

```
        Perm p = new Perm (pr,pc);


        p.setOriginal(P0);


        System.out.println("");



        PermArraylist list  = new PermArraylist();
        System.out.println("The initialised  candidate permutation matrix");
        p.printOriginal();
        System.out.println("");



        System.out.println("Level is  Starting");
        TempList = list.getPermsByLevel(pr, pc,0, P0);
         list  = new PermArraylist(TempList);

        for(int  j  = 1;j  < pr ;j++)
        {
         System.out.println("************************** this is level "+(j)  +"**********************");
         TempList2 = new ArrayList<Perm>();
         for  (int  i  = 0;  i  <list.getPermList().size();  i++)
            {

                TempList = list.getPermsByLevel(pr, pc,j,  list .getPermList().get(i).getOriginal());
                TempList2 = BigList(TempList2,TempList);

            }
         System.out.println("size  is"+TempList2.size());
          list  = new PermArraylist(TempList2);
        }
    }
}
```

Listing 6.6: **The construction of a permutation class for all candidate permutation matrices**

```
/*
 * To change this  license  header, choose License Headers in Project  Properties.
 * To change this  template  file ,  choose Tools | Templates
 * and open the template in the  editor.
 */
package Final_Ullmans;

/**
 *
 * @author Kellen
 */

public  class  Perm
{
    private  int  r  ,c  ;
```

```
private int perm [][] = new int[r][c];

private int original [][] = new int[r][c];

private int level;

/**
 *
 * @param r this is the row size of the matrix
 * @param c this is the column size of the matrix
 */
public Perm(int r, int c)
{
    this.r = r;
    this.c = c;

    perm = new int [r][c];
    original = new int [r][c];
}



/**
 * This is used to print the original Matrix
 */
public void printOriginal()
{
    for (int i = 0; i < r; i++)
    {
        for (int j = 0; j < c; j++)
        {
            System.out.print(original[i][j]+" ");

        }
        System.out.println("");

    }

}
/**
 * This is used to print the permutation Matrix
 */
 public void printPermutation()
{
    for (int i = 0; i < r; i++)
    {
        for (int j = 0; j < c; j++)
        {
            System.out.print(perm[i][j]+" ");

        }
        System.out.println("");

    }

}
```

```
 /**
  * This method is used to make a deep copy of a matrix
  *
  * @param matrix this is the matrix going to be copied
  * @param r this is the row size
  * @param c this is the column size
  * @return this return a deep copy
  */
 public int [][] deepCopy (int matrix[][],int r,int c)
{
    int M [][] = new int [r][c];
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            M[i][j] = matrix[i][j];

        }

    }
    return M;
}

 /**
  *
  * @param idcol
  * @param col
  * @param pr
  * @return
  */
 public static boolean IsIdCol(int[] idcol,int col,int pr)

{
    boolean flag = false;

    for (int i = 0; i < pr; i++)
    {
        if (idcol[i]==col)
        {
            flag = true;
        }

    }
    return flag;
}
 /**
  * this is used to create a permutation matrix from the original matrix
  *
  *
  * @param row – this is size of the rows in the permutation
  * @param col – this is the size of the columns in the permutations
  * @return – this will return a permutation matrix
  */

   public int [][] permutationz(int row ,int col)
{
    perm = this.getOriginal();
```

```
        for (int i = 0; i < c; i++)
        {
            if (i==col)
            {
                perm[row][i]=1;
            }else
            {
               perm[row][i]=0;
            }

        }
        return perm;
    }

    /**
     *
     * @return r - row size
     */
    public int getR() {
        return r;
    }
    /**
     *
     * @param r - set the row size
     */
    public void setR(int r) {
        this.r = r;
    }
    /**
     *
     * @return - return the column size
     */
    public int getC() {
        return c;
    }
    /**
     *
     * @param c - set the number of columns
     */
    public void setC(int c) {
        this.c = c;
    }
    /**
     *
     * @return a permutation matrix
     */
    public int [][] getPerm() {
        return perm;
    }
    /**
     *
     * @param perm - this will set the permutation matrix of the class
     */
    public void setPerm(int[][] perm) {
        this.perm = perm;
    }
```

```
    /**
     *
     * @return a the original matrix before permutation has taken place
     */
    public int [][] getOriginal() {
        return original;
    }
    /**
     *
     * @param original set the orginal matrix
     */
    public void setOriginal(int [][] original) {
        this.original = original;
    }
    /**
     *
     * @return the level being computed
     */
    public int getLevel() {
        return level;
    }
    /**
     *
     * @param level set the level which is being computed
     */
    public void setLevel(int level) {
        this.level = level;
    }
}
```

Listing 6.7: **Construction of a set of all candidate permutation matrices**

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Final_Ullmans;

import java.util.ArrayList;

/**
 *
 * @author Kellen
 */
public class PermArraylist
{
    private ArrayList<Perm> permList = new ArrayList<Perm>();
    private ArrayList<Perm> TempList = new ArrayList<Perm>();


    private int B [][] = {{0,1,0,0},      //Adjacency Matrix from the data graph
                          {1,0,1,1},
                          {0,1,0,0},
                          {0,1,0,0}};
```

```
 private int   Q [][] = {{0,0,1},
                         {0,0,1},      //adjacency Matrix from the pattern graph
                         {1,1,0}};

 public PermArraylist(ArrayList pL)
 {
     this.permList = pL;
 }

 public PermArraylist()
 {

 }

public  boolean IsIdCol(int[] idcol,int col,int pr)

 {
     boolean flag = false;

     for (int i = 0; i < pr; i++)
     {
         if(idcol[i]==col)
         {
            flag = true;
         }

     }
     return flag;
 }

 public int idCol(int matrix[][], int row,int col)
{
    for (int i = 0; i <col ; i++)
    {
        if(matrix[row][i]==1)
        {
            return i;
        }


    }
   return -1;
}

 public static boolean idRow (int matrix [][], int row,int col)
 {
     boolean flag = false;
     int count = 0;
     for (int i = 0; i <col; i++)
     {
         int val = matrix[row][i];

         if(val == 1)
         {
            count++;
         }
```

```
        }

        if (count<=1)
        {
            flag  = true;
        }

        return  flag;
    }

    public   ArrayList<Perm> getPermsByLevel(int r ,int c,int lvl,int   [][]   ori)
    {
        this .TempList.clear();
        boolean idRows[] = new boolean[r];
        int  idCols[]  = new int [r];

        Perm p = new Perm (r,c);

        int   [][]   permuatation = new int [r][c];
        p.setOriginal(ori);

        p.setLevel(lvl);
        Perm temp = new Perm (r,c);


        for  (int  i  = 0;  i  < r;  i++)
        {
            idRows[i]= idRow(ori,i,c);
        }

        for  (int  i  = 0;  i  < r;  i++)
        {
            if (idRows[i]==true)
            {
                idCols[i]=idCol(ori,i,c);
            }else
            {
                idCols[i]=-1;
            }

        }

        for  (int  i  = 0;  i  < idRows.length; i++) // this checks the  rows with the condition of  having one ,  '1'
             in  each row
        {
            System.out.println( "Is  this  row a ID row " + i  + "  :  "  + idRows[i]);
        }

        for  (int  i  = 0;  i  < idCols.length;  i++) // this  checks which column has at most '1' in  it
        {
            System.out.println("The row "+i+ " column : "+idCols[i]);


        }
```

```
for (int i = 0; i < c ; i++)
{
    temp = new Perm (r,c);
    temp.setOriginal(temp.deepCopy(ori, r, c));

    if (temp.getOriginal()[lvl][i]==1) // this checks if there is a one in the row
    {


        if (IsIdCol(idCols,i,r)==false) // this checks if a column corresponds to a selected row
        {


            temp.setOriginal(temp.deepCopy(ori, r, c));
            permuatation = temp.permutationz(lvl,i);
            if (lvl ==(r-1))
            {
                System.out.println("This is the candidate permutation matrix");
            }
            temp.printPermutation();
            if (lvl ==(r-1)) // this does the multiplication according to ullmann
            {
                int A [][] = permuatation;
                MatrixMulti MM = new MatrixMulti ();

                int AB [][] = MM.multipli(A, B);

                int AT [][] = MM.transposeMatrix(A);

                int Mat [][] = MM.multipli(AB, AT);

                System.out.println("This is the resulting candidate ajacency matrix");
                for (int j = 0; j < Mat.length; j++)
                {
                    for (int k = 0; k <Mat[0].length; k++)
                    {
                        System.out.print(Mat[j][k]+" ");
                    }
                    System.out.println("");
                }

                System.out.println("END");

                boolean C [][]= new boolean [Mat.length][Mat[0].length]; // this is used to as a
                    matching matrix between the expected candidate matrix and the produced query
                    matrix
                int count =0;

                for (int k = 0; k < Mat.length; k++)
                {
                    for (int j = 0; j < Mat[0].length; j++)
                    {
                        if (Mat[k][j]==Q[k][j]) // this is were comparison takes place
                        {
                            C[k][j] = true;
                            count++;
```

```
                        } else
                        {
                            C[k][j] = false;
                        }


                    }

                }

                    if (count == (Mat.length*Mat[0].length)) //this is where the results from matrix
                        C are deduced
                    {
                        System.out.println("Exact Match");
                    } else if (count > 0)
                    {
                        System.out.println("Partial Vertex Similarities");
                    } else {
                        System.out.println("Not Even Close");
                    }



                }
                Perm temp2 = temp;
                temp2.setPerm(temp.deepCopy(temp.permutationz(lvl,i),r,c));
                permList.add(temp2);
                TempList.add(temp2);
                System.out.println("");

            }

        }


    }
    return TempList;

}

public Perm getPerm (int i)
{
    return permList.get(i);
}

public  ArrayList<Perm> getPermList() {
    return permList;
}

public  ArrayList<Perm> getTempList() {
    return TempList;
}

public  void setPermList(ArrayList<Perm> permList) {
    this.permList = permList;
}
```

```
    public  void setTempList(ArrayList<Perm> permList) {
        this.TempList = permList;
    }

}
```

Listing 6.8: **Implementing the condition in equation (10) to get candidate adjacency matrices**

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Final_Ullmans;

/**
 *
 * @author Kellen
 */
public class MatrixMulti
{

    public MatrixMulti() {
    }

    /**
     * This method is used to multiply two matrices together n return their product
     *
     * @param A - this is the first matrix to be multiplied
     * @param B - this is the second matrix to be multiplied
     * @return - this will return matrix multiplication between A and B = AB
     */
    public int [][] multipli(int [][] A, int [][] B) {

        int aRows = A.length;
        int aColumns = A[0].length;
        int bRows = B.length;
        int bColumns = B[0].length;

        if (aColumns != bRows) {
            throw new IllegalArgumentException("A:Rows: " + aColumns + " did not match B:Columns " +
                bRows + ".");
        }

        int [][] C = new int[aRows][bColumns];
        for (int i = 0; i < aRows; i++) {
            for (int j = 0; j < bColumns; j++) {
                C[i][j] = 0;
            }
        }

        for (int i = 0; i < aRows; i++) { // aRow
            for (int j = 0; j < bColumns; j++) { // bColumn
                for (int k = 0; k < aColumns; k++) { // aColumn
```

```
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }

        return C;
    }
    /**
     * This method is used to take a matrix n return a transposed version of that matrix
     *
     * @param m this is the matrix that needs to be transposed
     * @return return a transposed version of matrix 'm'
     */
    public int [][] transposeMatrix(int [][] m){
        int [][] temp = new int[m[0].length][m.length];
        for (int i = 0; i < m.length; i++)
            for (int j = 0; j < m[0].length; j++)
                temp[j][i] = m[i][j];
        return temp;
    }

}
```