Applying graph theory concepts for analyzing BIM models based on IFC standards

(Anwenden von Graphentheorie Konzepten für Analyse von Gebäudemodellen basierend auf dem IFC Standard)

Master Thesis

Approved by the Faculty of Civil engineering of the Technische Universität Dresden

Written by
Ahmed Nahar

Master's study program ACCESS, Matriculation no. 4114378

Supervisor:
Prof. Dr.-Ing. Raimar J. Scherer

Tutor:
Dipl.-Ing. Ali Ismail MSc.

Dresden

Date of submission: 16.01.2017

Definition of the Master's Thesis

For Ahmed Ahmed Babkier Ahmed Nahar, Master's study program ACCESS,

Matriculation no. 4114378

Topic:

Applying graph theory concepts for analyzing BIM models based on IFC standards

(*German*: Anwenden von Graphentheorie Konzepten für die Analyse von Gebäudemodellen b asierend auf dem IFC Standard)

Building Information Models (BIM) contain a huge amount on information and complex relationships between the model elements. The Industry Foundation Classes (IFC) is the common ISO standard to exchange BIM models, it has complex EXPRESS data model schema (semantic object-oriented model which defines a lot of relationships between model elements, their properties, connection between elements, hierarchical structure of spaces and zones, etc.). The IFC models are saved as XML or STEP files, which makes is difficult to retrieve information. Converting IFC models to a graph network s can help to solve this complexity and offer a flexible way to analyze the relationships between the model entities.

Objective:

Study the potential of using graph databases and concepts of graph theory in order to explore, visualize and analyze the relationships inside BIM models and run complex queries for information retrieval.

Specific tasks:

1. Analyze the IFC EXPRESS data model and create a Meta graph model to represent the most important objects and the relationships between them and investigate which kind of Meta-Information could be adequately be represented by the relationships between the elements and for which graph theory could be employed to deduce them.
2. Convert IFC models into Graph networks/databases based on an IFC data model server and graph database tools.
3. Applying filters and information retrieval queries and topology analysis on different BIM models based on the generated graph database and verify the results.
4. Analysis and discussion of the results.

Abstract

Labeled property graph models are a suitable way for representing and describing the vast amount of information inside Building Information Models (BIMs). Where graph models can be automatically generated based on data extracted from Industry Foundation Classes (IFC) models.

The aim of this master thesis is to study the potential of using graph databases and the concepts of graph theory to manage, visualize and analyze the information inside BIMs. This can be done through for example through applying data retrieval queries, applying pathfinding algorithms and analyzing the complex relationships between the BIM model entities.

The IFC data model is used in the present study to build the graph database; therefore, a brief background on IFC data schema architecture is presented to build-up basic understanding on how IFC models are encoded. This is followed by an introduction to graph theory concepts and graph databases, with the aim to investigate the capabilities of graph models in representing complex and rich datasets such as BIM models.

The research proposes a generic approach to create two kinds of graph models: (1) IFC Meta Graph (IMG) model based on the IFC EXPRESS schema and (2) IFC Objects Graph (IOG) model based on STEP physical file format. The IFC EXPRESS schema is used as data source to generate the IMG model to graphically represent the IFC classes and the relationships. The IMG model expresses the classes and their attributes as nodes, and the relationships between them as edges. While, the IFC objects model files are used to build the IOG models, where IFC data are extracted through an IFC data model server and then imported into graph database management system (www.neo4j.com).

In both cases of graph modeling, there is a need for a third-party system to enable the data transmission process from the IFC schema/models to Neo4j database. Therefore, the IFC data model server www.ifcwebserver.org has been used to generate the graph query statements scripts and extract the IFC data in form of comma-separated values (CSV) data files for exportation into the Neo4j graph database.

This is following by application of graph queries and filters to visualize and analyze building information and to explore the capabilities of graph database in answering realistic questions in building projects particularly. And secondly, to provide a tool for a comparison study of different IFC standard releases or different IFC model versions, when a single graph database is used to storage various IFC models. Finally, the graph model is used for indoor navigation applications in buildings using spatial and geometric information extracted from the IFC model.

Keywords: Graph theory; Building Information Modeling BIM; Industry Foundation Classes IFC; Neo4j.

Declaration of independent work

I confirm that this assignment is my own work and that I have not sought or used unacceptable help of third parties to produce this work. I have fully referenced and used inverted commas for all text directly or indirectly quoted from a source. Any indirect quotations have been duly marked as such. However, all Ruby scripts within the present study have been fully written by my tutor Mr. Ali Ismail.

This work has not yet been submitted to another examination institution – neither in Germany nor outside Germany – neither in the same nor in a similar way and has not yet been published.

Dresden, 16th January 2017

……

Ahmed Ahmed Babiker Ahmed Nahar

Acknowledgements

I would like to extend my heartfelt gratitude to my supervisor Prof. Dr.-Ing. Raimar J. Scherer, my tutor. Ali Ismail MSc. and Yaseen Srewil MSc., for their support through the learning process of the present master thesis.

I would like also to express my gratitude to my parents Najat and Ahmed, and to my brothers and sisters for their spiritual support. I would like also to thank my friends Ahmed Isam, Khideer Bashir, Ahmed Hamza, Salma, Khalid, Ammar, Aamir, Ahmed Sultan and Nyazi for their helpful support. Finally, this work is dedicated to the soul of my brother Babiker.

Dresden, 16th January 2017

Ahmed Ahmed Babiker Ahmed Nahar

Table of Contents

List of Figures

List of Tables

List of Listings

Notation and Abbreviation

AEC	Architecture, Engineering, and Construction
AEC	Architecture, Engineering, Construction
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BIM	Building Information Modelling
BIMs	Building Information Models
CSV	Comma Separated Values
DB	Database
DML	Data Manipulation Language
DSL	Domain Specific language
GPL	General Public License
IOG	IFC Object Graph
IFC	Industry Foundation Classes
IFCID	Industry Foundation Class Identifier *similar to* STEP ID
IMG	IFC Meta Graph
RDF	Resource Description Framework
SDK	Software Development Kit
SMC	Solibri Model Checker software
SPF	STEP Physical File
STEP	Standard for the Exchange of Product data
STEP ID	Unique identifier for instances within STEP Physical File
URI	Uniform Resource Identifier
XML	Extensible Markup Language
2D	Two-dimensional
3D	Three-dimensional

Typographical conventions

The following typographical conventions are used in this thesis:

Italic	to indicate the IFC classes within the text.
Upper case	to indicate Cypher commands within the text.
Courier new font	used for program listings to indicate Cypher commands.

Chapter 1: Introduction

1.1 Motivation

The very fast development in the sector of information technology has been successfully exploited in the design and construction fields to develop useful concepts and applications such as Building Information Modeling (BIM), and to provide tools and software capable of meeting the users' requirements for better digital representations and data management of the building projects.

This step forward in BIM technologies had come with the challenge of dealing with huge amount of data and building information, which could remain inaccessible in several cases due to absent of suitable data management approaches inside BIM tools. For instance, when the Industry Foundation Classes (IFC) data model is used as a standard and natural data exchange format for Building Information Models (BIMs), the relations among IFC entities could remain implicit and unused in several cases.

Whereas, conversion of BIMs into an effective information retrievable model could significantly facilitate the efforts of exploring and analyzing highly connected data, such as buildings information within IFC object model. Evidently, graphs have shown great capabilities in/for understanding complex and rich datasets in many different fields of the life. Therefore, IFC object graphs that represent entities as nodes and relationships could be a useful tool for data management in construction and engineering.

1.2 Aims and objectives

Graph models are extremely useful for representation and description of the complex relationships among building elements and data within BIMs [1]. Thus, transferring of such connected data as in the case of IFC data model into a semantic graph database can significantly support the processes of retrieving building information and execution of queries.

Therefore, the present master thesis will, firstly, study the potential of using graph database and the concepts of graph theory to explore, visualize and analyze BIM models, and run complex queries for information retrieval. And secondly, Complex queries will be performed on the generated graph models, to develop set of graph-based queries for frequently executed search. Thus, the pre-defined graph-based queries can facilitate application of queries for a user with trivial skills. Finally, a simple example of building topology analysis is carried out to check the capability of graphs to provide path-finding algorithms for indoor navigations routes.

1.3 Layout of the thesis

To achieve the objectives of this study, the general structure of the thesis has been divided into six main chapters, in an attempt to present the topics in a logical order as depicted in the work-flow of Figure 1.1 below, and summarized in the following paragraphs:

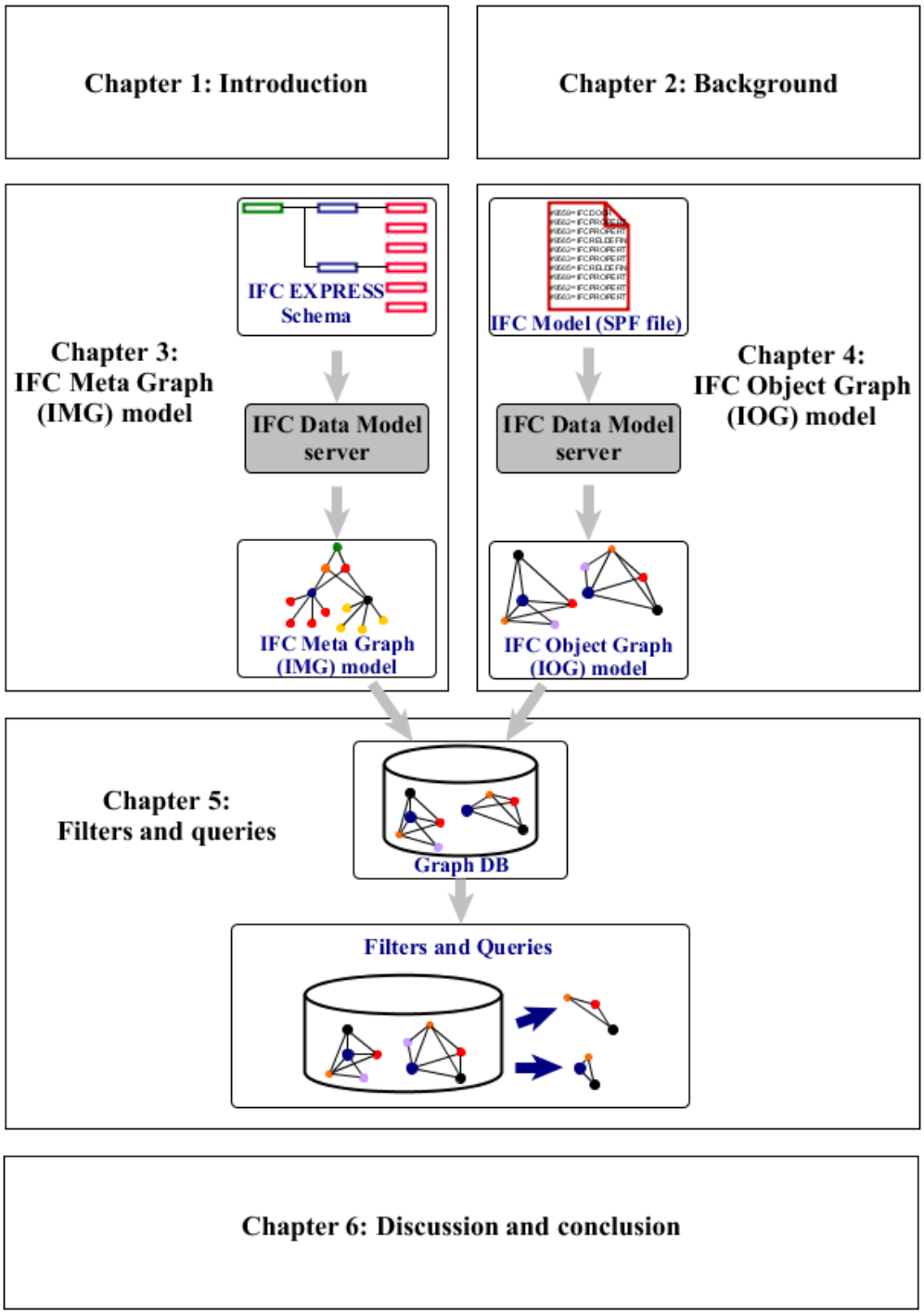

Figure 1.1: General structure of the thesis

Chapter 1 states the motivations and utilization of graphs to diagrammatically manage and represent building information within BIMs, by using IFC models as a data source to develop labeled property graphs database.

Chapter 2 presents an intuitive background on IFC architecture and IFC EXPRESS data model in general. This is followed by an introduction to the state of the art in graph theory and graph database management systems.

Chapter 3 illustrates a generic approach to generate IFC Meta Graph (IMG) models based on IFC EXPRESS schema, with the aim to graphically represent the IFC classes and the relationships among them as labeled nodes and generic relationships. Then, the IMG models are parsed by applying filters and queries for information retrieval.

Chapter 4 describes two approaches to create IFC Object Graph (IOG) models using IFC data models in STEP physical file format. The manual approach is introduced in advance, to illustrate the data transmission process in detail, while the automatic approach provides a practical methodology for generating graph database. Additionally, the idea of using single graph database to store multiple IOG models or graphs enhancement by having nodes with multiple nodes is demonstrated.

Chapter 5 demonstrates a verification process to examine the correctness of the IOG models that generated in Chapter 4, and then, evaluates the capability of theses graphs in retrieving building information to provide realistic answers for frequently applied queries. Finally, advanced analysis of BIMs in the area of building topology to provide indoor navigation routes is presented as an example for IOG graphs' utilizations.

Chapter 6 discusses the potential of using graph concept to visualize and analyze IFC models based on the queries and filters applications and addresses recommendations for future research.

Chapter 2: Background

2.1 Industry Foundation Classes

2.1.1 Introduction

Industry Foundation Classes (IFC) is a data model developed by buildingSMART International [2] to support the exchange of building information in Architecture, Engineering and Construction (AEC) industry to improve collaboration, scheduling, cost and delivery time activities throughout the whole life cycle of the building project. This enables actors from different disciplines to effectively use BIM data and to fully exchange information in construction or facility management projects, as shown in Figure 2.1 below.

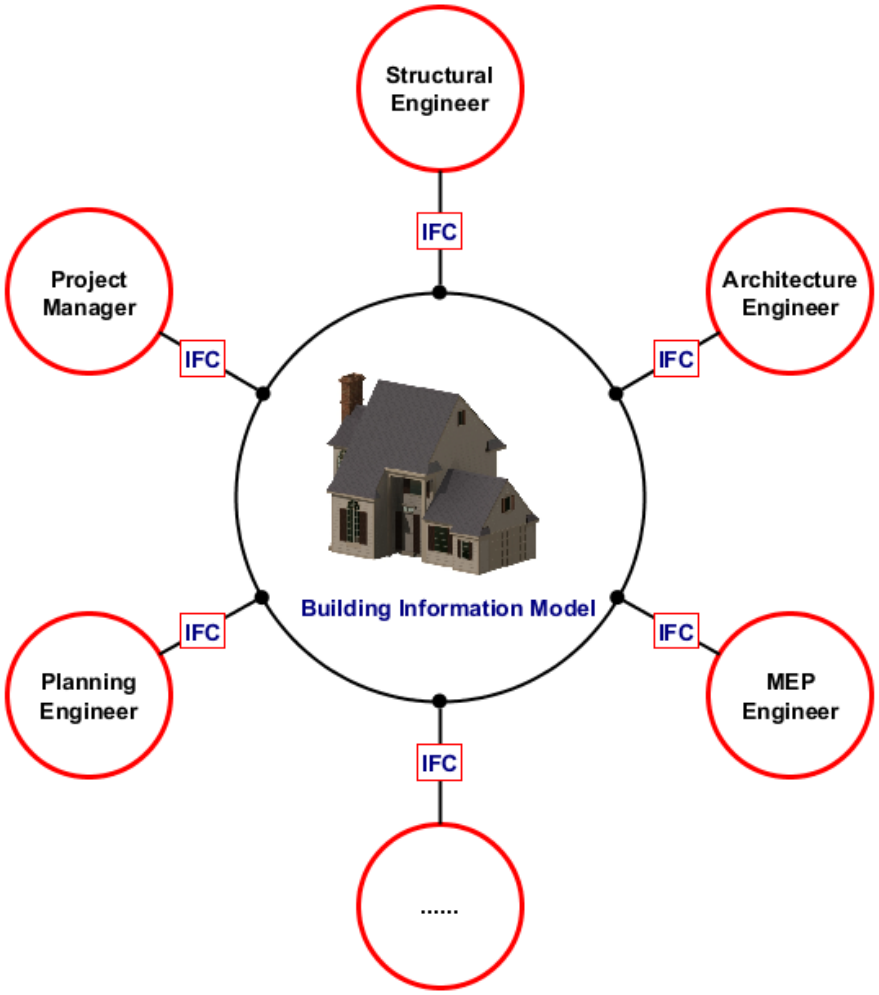

Figure 2.1: Use of IFC as standard BIM data exchange format

However, the IFC schema is described using EXPRESS specification language that defined by ISO 10303-11, as conceptual information modeling language. The specification specified also a

graphical representation known as EXPRESS-G to provide graphical subset. The IFC data schema can be saved as Extensible Markup Language (XML) file structure per ISO10303-21 or STEP Physical file (SPF) document structure following ISO10303-28, and having file extension '.ifc'. The basic structure of the SPF file is divided into two main sections, which are the header section and the data section [3]. The header section represents general information regarding the file itself such as file version, the application used to export the file, date and time of exportation, and the name of company or person own the file. While, the data section contains the instances of the entities based on the IFC specification, entity type name, and a list of attributes. The excerpt in Listing 2.1 below, is showing an example of SPF basic structure with header and data sections.

```
HEADER;
FILE_DESCRIPTION(('ViewDefinition [CoordinationView]'),'2;1');
FILE_NAME('D:/wmseclipse/IfcRouting/./data/Test/Muster003.ifc','2014-03-
18T09:48:12+0100',(''),(''),'IFC Tools Project - IFC2x3 Java
Toolbox','20130308_1515(x64) - Exporter 2014.0.2013.0308 - Default UI','');
FILE_SCHEMA(('IFC2x3'));
ENDSEC;

DATA;
#4= IFCPERSON($,'Undefined',$,$,$,$,$,$);
#6= IFCORGANIZATION($,'Undefined',$,$,$);
#10= IFCPERSONANDORGANIZATION(#4,#6,$);
#13= IFCORGANIZATION('GS','Graphisoft','Graphisoft',$,$);
#14= IFCAPPLICATION(#13,'1.0.0','ArchiCAD-64','IFC2x3 add-on version: 3006
ITA FULL');
#15= IFCOWNERHISTORY(#10,#14,$,.ADDED.,$,$,$,1425638738);
#16= IFCSIUNIT(*,.LENGTHUNIT.,.MILLI.,.METRE.);
#17= IFCSIUNIT(*,.AREAUNIT.,$,.SQUARE_METRE.);
#18= IFCSIUNIT(*,.VOLUMEUNIT.,$,.CUBIC_METRE.);
..
..
..
..
..
..
..
ENDSEC;
END-ISO-10303-21;
```

Listing 2.1: Basic structure of SPF file with header and data section

Currently, there are several IFC-based software tools provided for visualization of geometric and properties content of the IFC data. Such as the IFC engine DLL [4] that developed by RDF or the Solibri Model Checker for analyzing BIMs. However, most of these tools are built for specific preliminary task or study and they are not provided with API to enable users with different interests to make use of them [5].

2.1.2 IFC Architecture

Each IFC model is composed of IFC entities built up in a hierarchical order, where each IFC entity includes a fixed number of IFC attributes, plus any number of additional IFC properties. The IFC attributes are the main identifiers of the entities, while the names of these attributes are fixed, having been defined by buildingSMART [2] as part of the IFC standard code.

The IFC data schema architecture highly depends on four conceptual layers to define the IFC entities within a Building Information Model. buildingSMART [2] defines these conceptual layers as Resource layer, Core layer, Interoperability layer and Domain layer, as shown in Figure 2.3. In fact, entities that represented within the second, third and fourth layers are subtyped or derived from the root class *IfcRoot*, either they referenced directly or indirectly to the *IfcRoot*. The remaining entities which are non-IfcRoot instances belong to Resource layer. Thus, the *IfcRoot* class is considered as one of the super abstract entity types which contain three fundamental entity types in the IFC model as shown in Figure 2.2. These entity types are the first level structure of the *IfcRoot* entities hierarchical.

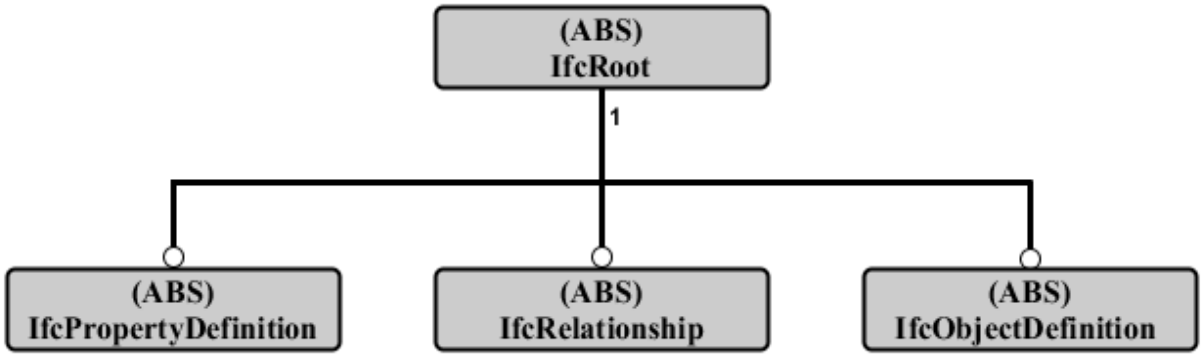

Figure 2.2: Fundamental entity types derived from *IfcRoot* class

i. IfcPropertyDefinition

The *IfcPropertyDefinition* type describes all characteristics that may attach to objects. Thus, valuable information can be shared among multiple object instances. However, it may express the occurrence information of the actual object in the project context, in case that it is attached to a single object instance.

ii. IfcRelationship

IfcRelationship summarizes all the relationships among objects. This can enable users saving relationship specific properties directly at the relationship object and avoid duplication of relationship semantics from the object attributes.

iii. IfcObjectDefinition

IfcObjectDefinition stands for all handled objects or process. Where, all physical items and products such as roofs, windows, and slabs that can be touched and seen are classified as *IfcObjectDefinition*.

However, Li [6] developed an illustration model to describe the subtype entities of *IfcRoot* class, with respect to a real building model, and illustrates the distribution of these entities within the conceptual layers, in order to provide an object-relational storage model for IFC data.

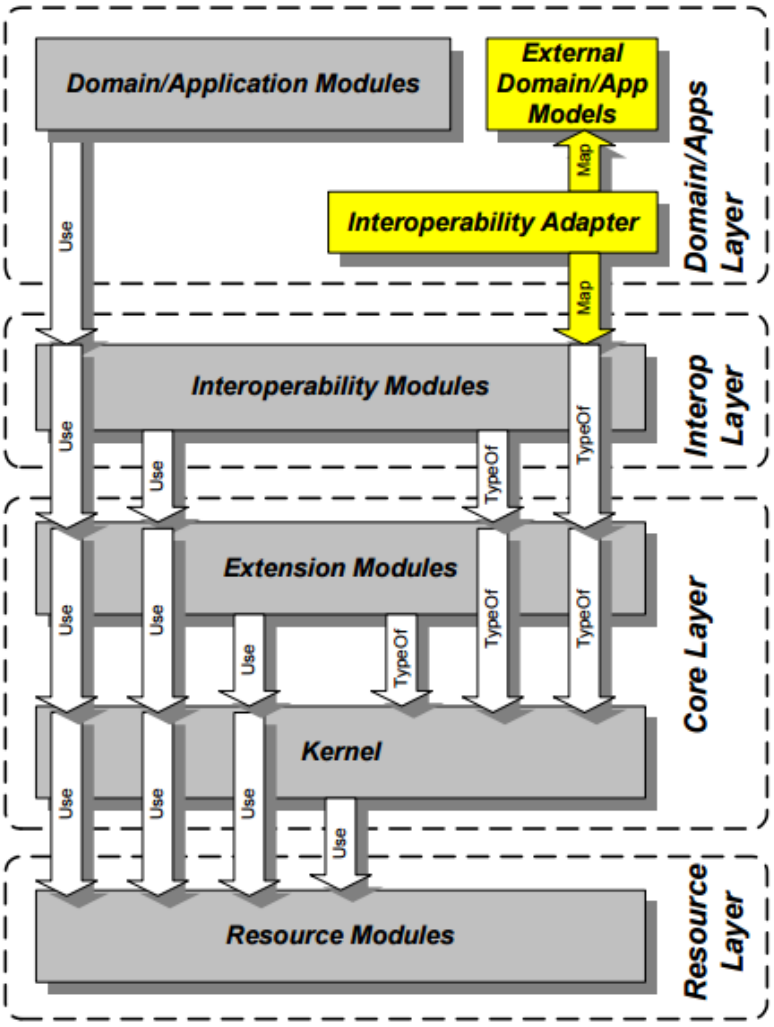

Figure 2.3: IFC conceptual layers

The present section starts out by introducing the IFC entities and their conceptual layers. In the following part, the connectivity among these entities is examined. Where, the abstract objectified relationship *IfcRelationship* and its subtype relationships are responsible for connectivity among objects, in which several properties can be attached to each relationship. However, this concept of the objectified relationships enables the generation of a separate subtype tree for relationship

semantics. In fact, there are two different types of relationships exist within the subtypes of *IfcRelationship*:

1. One-to-one relationships.
2. One-to-many relationships.

The following Figure 2.3 demonstrates the different types of objectified relationship within the IFC2x3 and their abstract supertype *IfcRelationship*:

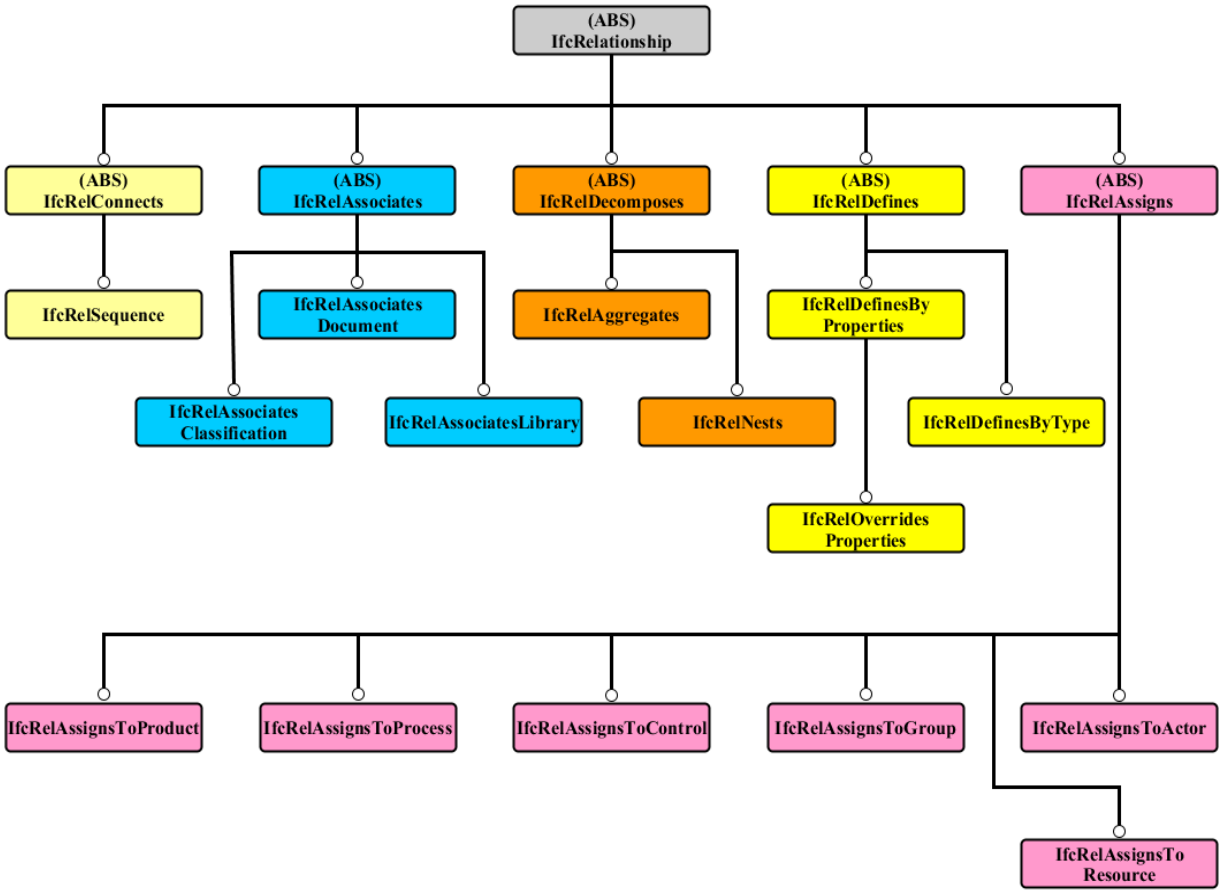

Figure 2.4: IFC abstract objectified relationship

2.1.3 IFC EXPRESS data model

Given that the IFC is used as an information database for the work presented in this study. Therefore, understanding of how data are represented within the IFC schema file and their possible relationships, is highly required.

In fact, there are two different kinds of classes exist in the IFC data schema model. These two kinds are the entity classes and type classes. The entity classes represent all object types contained in a certain model. While the type classes describe data types of that model. For instance, all the window instances in a certain model are members of the group window class, but each member of this group is unique and can be sorted out from other windows.

IFC classes have attributes attached to them which could indicate a relationship to another object or they could just be attached as simple data type attribute, e.g. integer, string, logical, or boolean. Therefore, IFC models distinguish between the attributes that are directly attached to object as entity attributes, and attributes that assigned to indicate a relationship to other objects. The second of two attributes are the most adopted approach to extend viable properties. However, as it is depicted in Figure 2.5 below, each IFC class could have simple data attributes with referenced object attributes as in the case of the *IfcRoot*, or referenced object with relationship attributes as in the case of *IfcProduct*, or entity attributes and relationship attributes together as in the case of *IfcObject*.

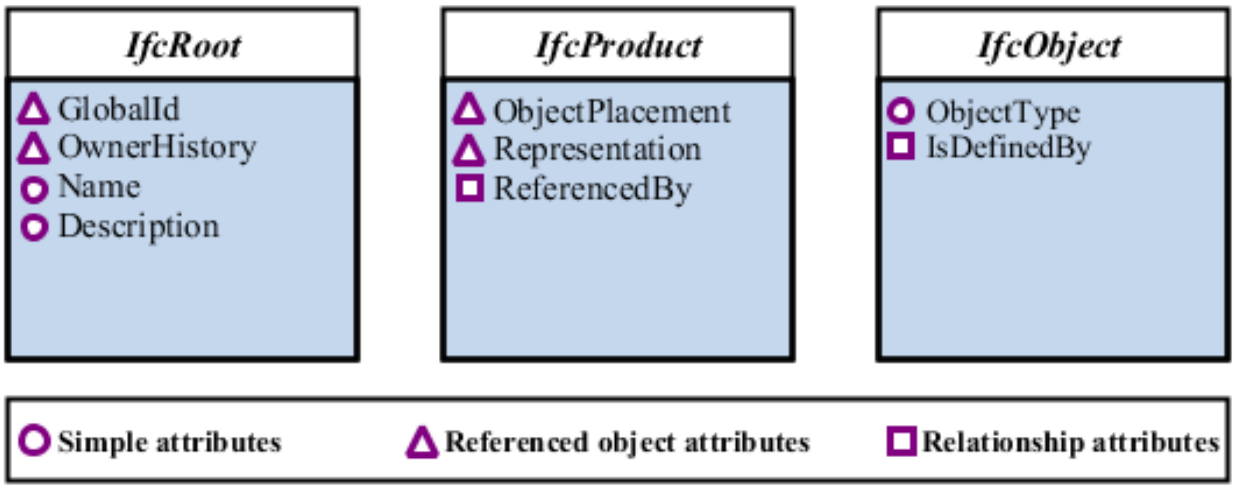

Figure 2.5: Entity attributes and relationship attributes

In addition to the direct attached attributes, the objects can inherit attributes from their supertype classes, where, property pair can be assigned to almost any type of objects and thus to support enlarging their attributes base.

2.2 Graph theory

2.2.1 Introduction

Graph theory is the branch of mathematics that dealing with the study of graphs [7]. While the concept of graphs itself is defined by graph theory as a diagrammatic representation of real-world scenarios in a form of points and lines [8]. The points are called vertices or nodes, while the lines that connect them are so-called edges, arcs or relationships. Each vertex in the graph is represented by drawing a dot or a circle, while the relationship between each pair of vertices is indicated by drawing an arc or line if they are connected by an edge, as it is shown in Figure 2.6. In such diagram, the relative position of the nodes does not play an important role in the data representation of the graph [9].

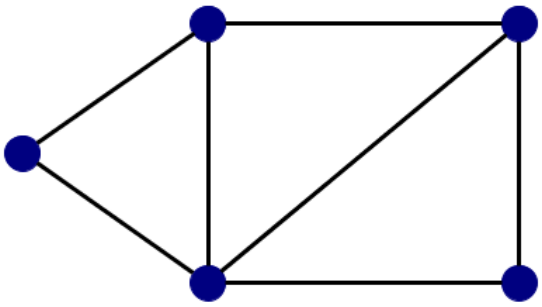

Figure 2.6: Graph consists of nodes and edges

A graph is called simple graph if it consists of unique edges connecting each pair of nodes, as has been presented in Figure 2.6 above. However, the concept of simple graphs can be extended to another type of graphs in which several edges are joining the same pair of vertices, or a vertex joining to itself by a loop edge, as shown in Figure 2.7 below. The resulting graph, in which multiple edges and loop edges are used, is called general graph. However, these concepts about simple graphs and graphs, in general, will be discussed in detail in the following sections.

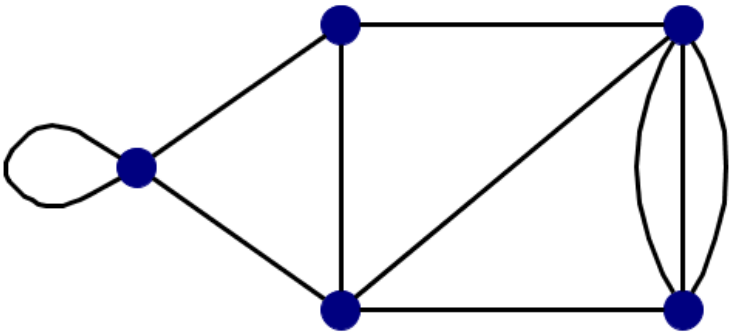

Figure 2.7: Graph with multiple edges and loop edges

2.2.2 History of graph theory

Mathematicians consider the article that wrote in 1736 by Leonhard Euler [10] regarding the problem of Seven Bridges of Königsberg as the first study in the history of graph theory and true

proof in the theory of networks. The issue is that the citizens of Königsberg used to amuse themselves by rambling through the city, visiting all parts of the city with an attempt to cross each of the seven bridges exactly once. However, the old city of Königsberg is separated into four parts; two main lands and two large islands, by the River Pregel, as it can be seen in Figure 2.8. These parts were connected by seven bridges to allow citizens to move from one part of the city to another. Therefore, Euler studied the possibility of finding a route for visiting all parts of the city by crossing each of the seven bridges just once.

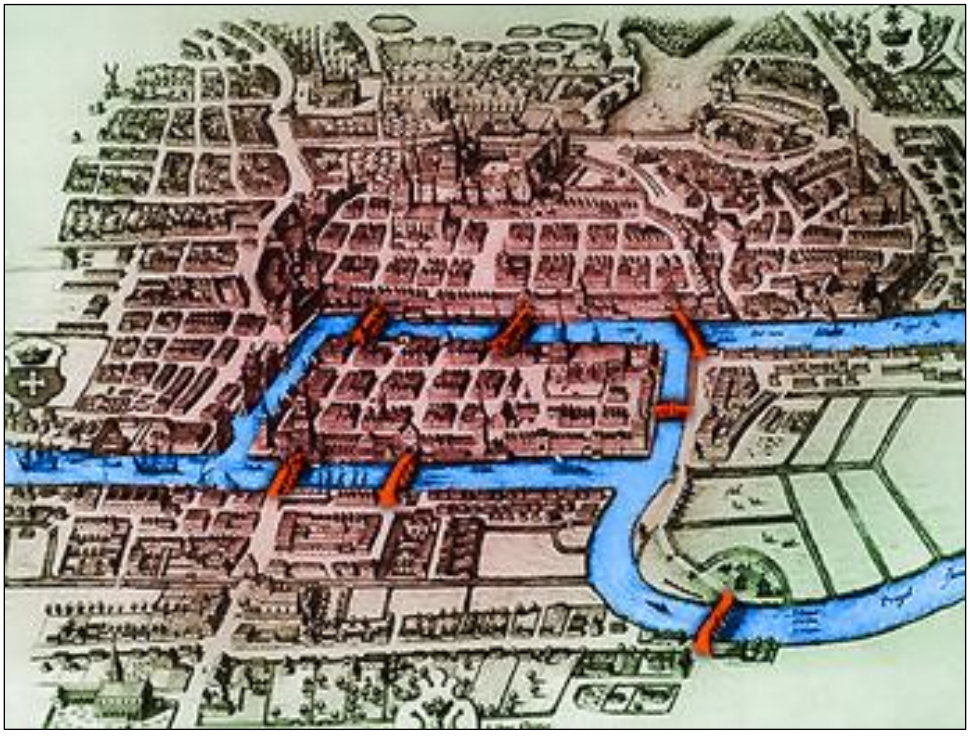

Figure 2.8: A map of Königsberg showing the seven bridges

Later on, and in the middle of the nineteenth, Kirchhoff and Cayley have established the practical foundations of graph theory in the physical world with different implementations cases. In 1847, Kirchhoff [11] developed the theory of trees in order to provide solutions for the system of simultaneous linear equations to indicate the current and voltage, in each part and around each circuit of an electric network. While, Cayley [12] relied on differential calculus to study the potential of using a specific class of graphs which is a tree, in order to find the number of isomers in saturated hydrocarbons based on the number of carbon atoms, as shown in Figure 2.9.

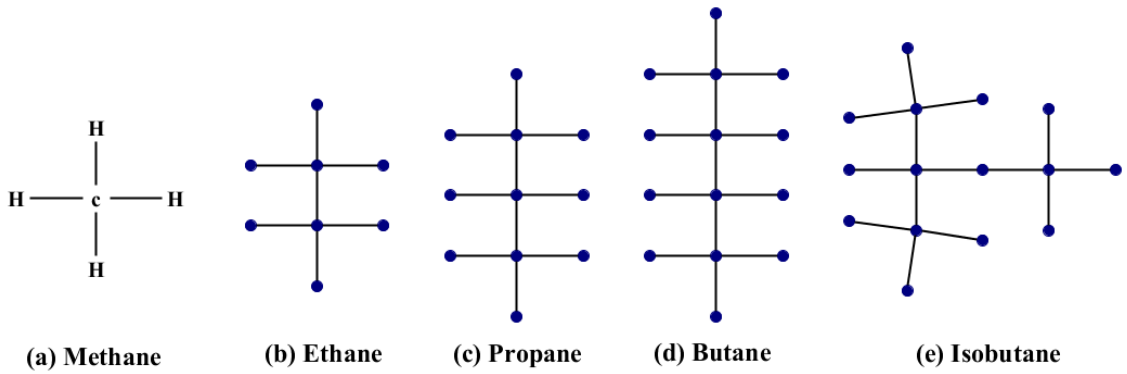

Figure 2.9: The smallest saturated hydrocarbons

The textbook that written by Hungarian graph man Dénes Kőnig and published in 1936, is the first reference book on graph theory [13]. However, a second book was published in 1969 by Frank Harary [11] in an attempt to represent most of the graph theorems to enabled mathematicians and graph users from other disciplines to come together. Currently, several textbooks are available as reference books to introduce the concept of graph theory [10] [9] [8].

In 1852, Francis Guthrie [11] has introduced one of the most remarkable problems in the history of graph theory which is a so-called four-color problem, to enable separation of a plane into tangency regions, where each two adjacent regions could have different colors. This problem has remained unsolved for a long time where several erroneous proofs have been proposed. The first of these proposals was given in 1879 by Kempe [14]. Then an error was discovered in 1890 by Heawood, in fact, Heawood proposed using five colors instead of four colors to overcomes the errors of Kempe's proposal. Eventually, the four-color problem was settled by K. Appel and W. Haken [8] in 1976 as in an alternative formulation, by making use of Heesch's method for solving the four-color problem using computers.

The second remarkable milestone in the history of graph theory was introduced by W. Hamilton [9] in 1859. In fact, Hamilton invented a mathematical game based on a regular solid dodecahedron with 12 faces and 20 corners as vertices. In the game, the player is challenged to pass through the vertices and find a circuit along the edges which passes through each vertex exactly once.

The twentieth century has witnessed the implementation of graph theory concepts in other areas. Practical examples of graph use come from the work of the psychologist Lewin [11]. Lewin has proposed in 1936 that the daily activities and life of any person can be represented by a planar map, where the map regions represent the various activities such as working environment, outdoor activities, and hobbies. Then, the map is converted into a graph as shown in Figure 2.10.

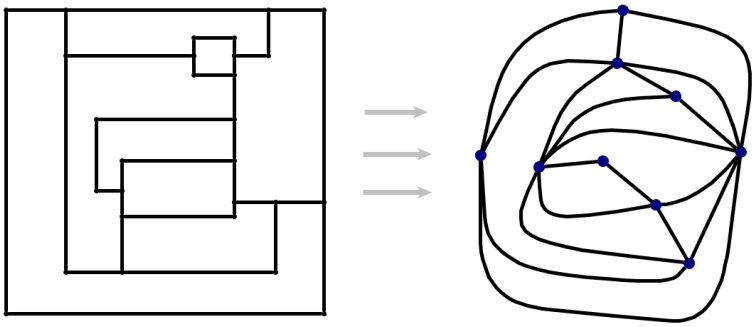

Figure 2.10: Activities map and its corresponding graph

From the beginning of this century, graph theory has gone through a noticeable change in the way of sorting data sets and examination of interrelations network, indicating a new phase in the history of graph theory. It is a long journey, from the "seven bridges of Königsberg" problem to the uses of spectral methods and sophisticated combinatorial in the fields of mathematics and computer science.

2.2.3 Graph data structures

Although it is appropriate to display a graph using a diagram of nodes and lines. However, such method may not be a suitable approach when it comes to computer processing and storing the graph in a computer system [8]. Therefore, data structures such as list and matrix structures are often used to organize and store data in a computer. There are two common cases available for graph representation in the case of using list structure; the incidence list where the vertices are listed in an array of pairs, and the other case is an adjacent list where a collection of the neighboring vertices or edges are stored separately. An example of a simple graph $G = (V, E)$ and its adjacent list is shown in Figure 2.11.

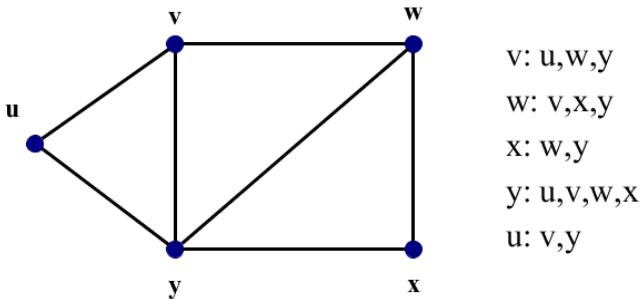

Figure 2.11: Simple graph and its corresponding adjacent list

Similar to the list structure, the matrix structure has two approaches for representing graphs [15]. For instance, a graph $G = (V, E)$ with n number of vertices can be represented by a n x n adjacency matrix (A), whose A_{uv} entry is the number of edges joining vertex u and vertex v. Alternatively, the graph $G = (V, E)$ can also be represented by an incidence matrix (M). In this case, if $A_{uv} = 1$, then

vertex *u* is incident to edge *v*. While $A_{uv} = 0$ indicates non-adjacent objects. Figure 2.12 shows a labeled graph *G* with its adjacency and incidence matrices.

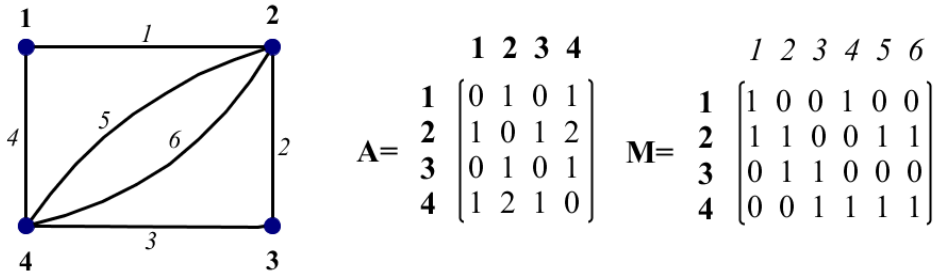

Figure 2.12: Labelled graph, and its adjacency and incidence matrices

The list structures are preferable to matrix structures since they require less memory. On the other hand, matrix structures provide better performance in terms of data accessing although they require huge memory. However, a combination of both structures provides a better methodology for realistic applications.

2.2.4 Definitions and fundamental concepts

In this section, we provide several basic definitions and fundamental concepts regarding graph theory. This intuitive background on graph theory will be used more formally and practically in the later chapters. As mentioned earlier, graphs are considered as an important technique to describe relational structures. Where, each graph *G = (V, E)* consists of a finite set *V(G)* of vertices and a finite multiset *E(G)* of edges with unorder pair (*u*, *v*) of vertices. The edge is called self-loop or loop edge when *u = v*, which means that the vertex is joining to itself. In the case, where several edges are joining the same pair of vertices, then *E* is a multiple edge. Thus, an edge {*u*, *v*} is said to join the vertices *u* and *v*. For instance, Figure 2.13 represents a simple graph *G* whose set of vertices *V(G)* is {*v*, *u*, *z*, *w*}, and whose edge set *E(G)* comprises of the following edges *vu*, *vz*, *vw and zu*.

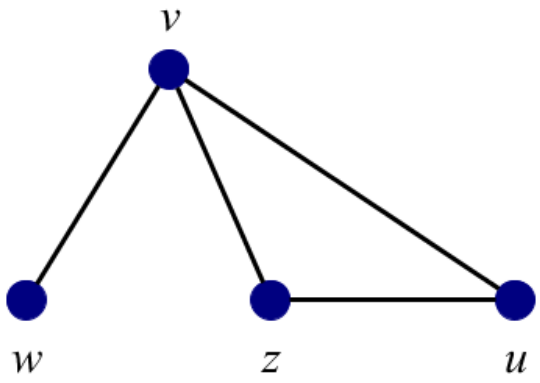

Figure 2.13: Simple graph with finite sets of vertices and edges

2.2.4.1 Plane and Planar graph

In graph theory, a graph *G = (V, E)* is called planar graph if its edges can geometrically be represented without intersection except at vertices, such graph can easily be redrawn in a plane without crossings [16] among its edges. The geometric drawing of the planar graph *G* is called a planar embedding of the graph, while other mathematicians [9] refer to the planar embedding of a planar graph as a plane graph. Figure 2.14 illustrates a planar graph G and its planar embedding.

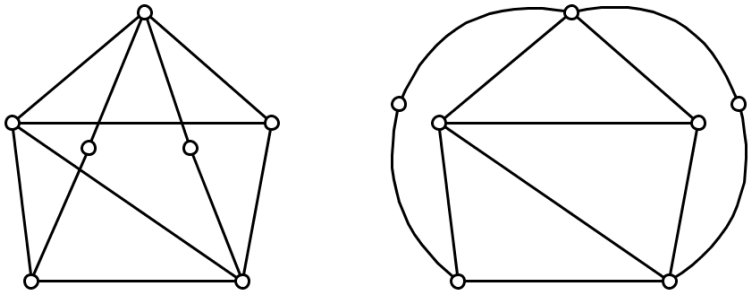

Figure 2.14: (a) Planar graph G and its (b) planar embedding **[9]**

Each connected part within the plane is called a region if it is surrounded by edges and does not contain and vertices. Furthermore, the region that lays outside the embedding is called the exterior region. However, deciding whether a graph is planar or not, has several important advantages for many practical situations. For example, specifying whether a certain electrical network is planar or not, is highly important for representing circuits layouts.

2.2.4.2 Undirected and Directed graphs

A graph *G = (V, E)* with un-oriented edges is called undirected graph [16]. This kind of graphs is used when there is no need to point out the direction of the relationships, which means that the edge from vertex *u* to vertex *v* can be written as (*u, v*) and or (v, u) as depicted in Figure 2.15. However, when graph's edges are oriented, the resulting graph *G = (V, E)* is called directed graph or digraph. In this case, the arc from vertex *u* to vertex *v* is written as (*u, v*) while the other pair (*v, u*) indicates the opposite direction arc. In some special cases, graph users may combine some directed edges with undirected edges in the same graph, in such case the graph is a so-called mixed graph.

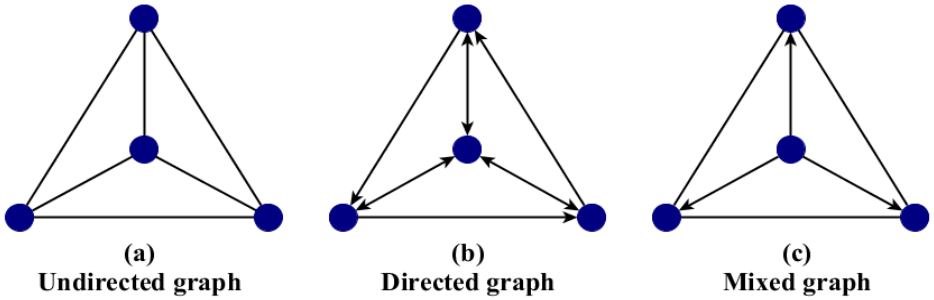

Figure 2.15: Directed and undirected graphs

2.2.4.3 Labeled property graphs and Isomorphism

Nowadays, the application of graphs has become an important technique to describe several scenarios in the real-world. One of the applications of graphs is to provide a simplified description of scenarios datasets in a way that produce a useful understanding of a complicated data. This has led to the birth of a special form of graph model, the so-called labeled property graph [17]. Labeled property graphs are similar to simple graphs, consist of nodes and relationships which are often expressed as vertices and edges as depicted in Figure 2.16. However, labeled property graphs provide additional characteristics to facilitate graph understanding, where, nodes could have a single or multiple labels, in addition, they could have properties (key-value pairs). Relationships can also be named and contain properties while connecting each two nodes as start and end node.

Obviously, all the nodes within Figure 2.16 are labeled (Book, Publisher, and Reader), to indicate their role in the network. These nodes are then connected using titular relationships to further clarify the expressive structure. Both nodes and relationships have key-value properties except for ‘read’ relationship. For instance, the relationship ‘published_by’ has two key-value properties which are ‘Year: 1993’ and ‘Edition: Fourth edition’. This indicates that the fourth edition of the book ‘Graph theory’ has been published by ‘Dover’ in 1993.

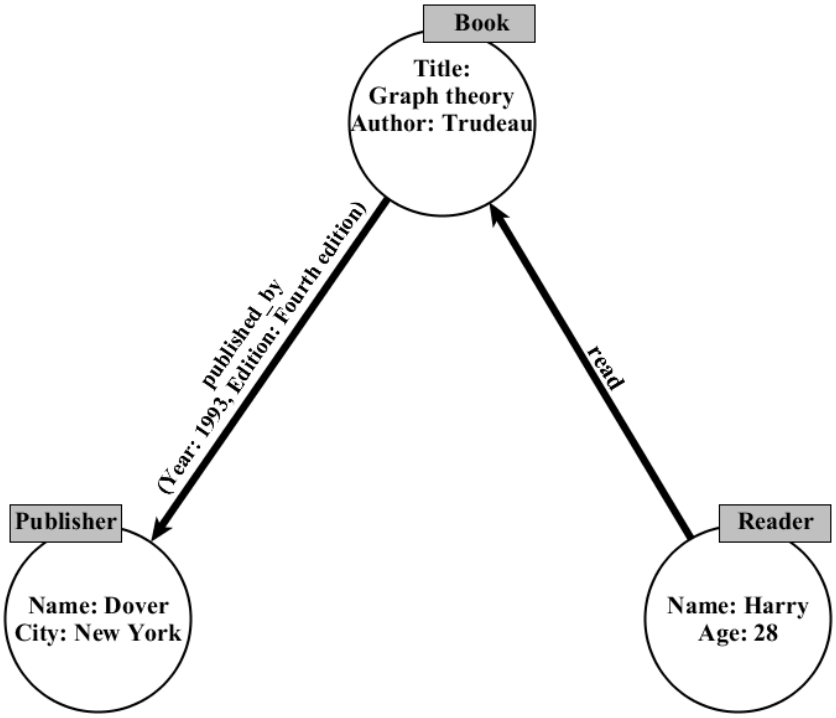

Figure 2.16: Labeled property graph

Furthermore, two graphs G_1 and G_2 are called isomorphic and written $G_1 = G_2$, if they are identical in which $V(G_1) = V(G_2)$ and $E(G_1) = E(G_2)$ [16], which means that there is a one-to-one correspondence between their objects. For instance, G_1 and G_2 of Figure 2.17 are isomorphic if all the vertices (v_i) of G_1 are identical with their correspondence vertices (u_i) from G_2. The isomorphism problem is an important research problem in graph theory for networks analysis to know if two graphs are isomorphic or not.

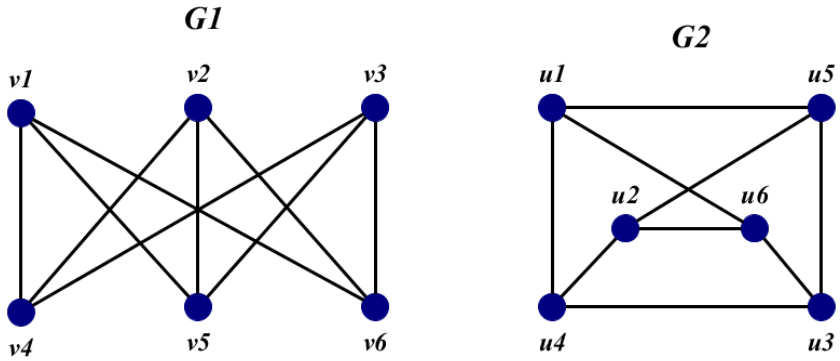

Figure 2.17: Isomorphic graphs

2.2.4.4 Paths and Connectivity

In this section, we present two of the most elementary properties of graphs, which are the path and the connectivity. Firstly, for any graph $G = (V, E)$, the terminology "Walk" is used to describe any

finite sequence of alternating vertices and edges [11] which exist in a form of (v_0, e_1, v_1, e_2…, e_n, v_n). The first vertex v_0 in the walk is called "Initial vertex", while the last vertex v_n is called "Terminal vertex". Here it is important to mention that a walk is called a "Closed walk" if the initial vertex of the walk and the terminal vertex are the same vertex, but when the walk starts from the specific vertex and ends at another vertex then it is considered as "Open walk". However, this brief introduction to the term "Walk" was given here as a preface to the term "Path". Thus, a walk is a path if all the alternating vertices and edges along the walk, are traversed only once.

Moreover, graph edges can be associated with labels, as it has been illustrated in the previous section. However, if an edge is labeled by a real number then is called weight, and graph *G* with weights associated with its edges is called "Weighted graph". Weighted graphs support many applications and problems in graph theory, such as the shortest path problem [10]. For example, if a weighted graph representing a road network as it is shown in Figure 2.18 below, the weights represent the length of each road. To determine the shortest route between the two towns *v1* and *v2*, one must find the path that achieves the minimum weight connecting these two towns, as it is shown in Figure 2.18 by thick lines.

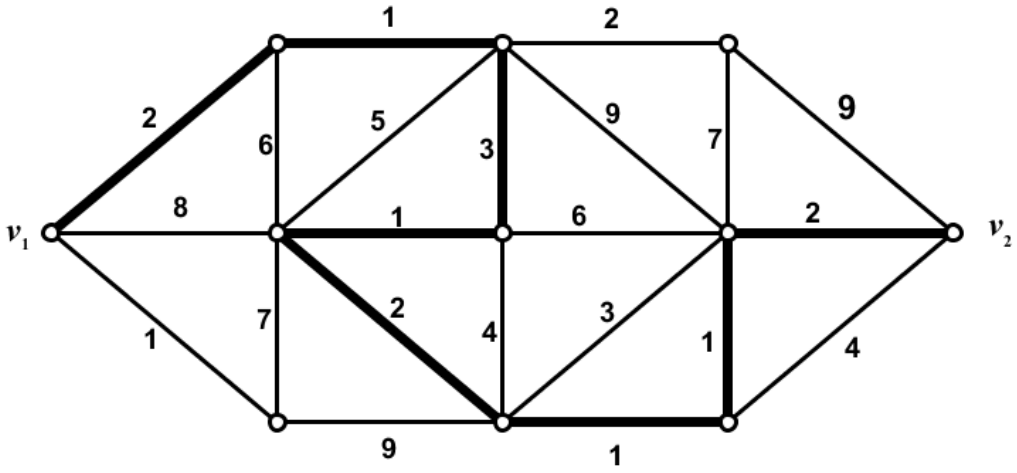

Figure 2.18: Path of minimum weight

The second concept within this section is regarding "Graph connectivity", which is one of the most significant characteristics in the graphs study. Thus, a graph is defined as a connected graph only if there is a path between each pair of its vertices [8]. Therefore, the connectivity rate of each graph depends on the number of edges that need to be removed in an aim to have a disconnected graph. Thus, the tree graph is the minimally connected graph, since it can be disconnected by deleting single edge. Nevertheless, rooted trees are the most used type of graphs in the case of hierarchical data such as pedigree chart or organization structure [18]. Figure 2.19 displays an example of a simply rooted tree, in which, vertices can be represented per their level from the root.

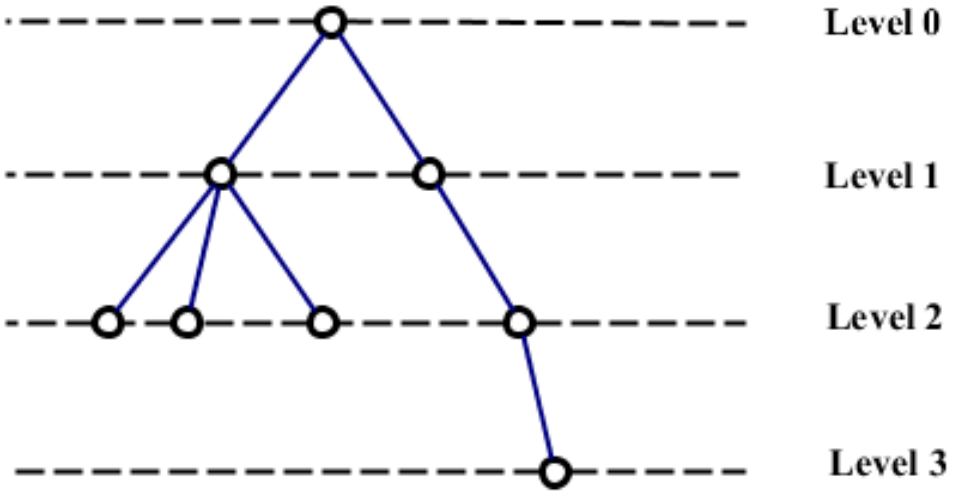

Figure 2.19: Representation of rooted tree

2.3 Graph data management

2.3.1 Introduction

Several graph processing systems have been developed in the last decade to meet the modern graph modeling and analysis tasks. Doekemeijer [19] has declared that more than 80 systems have been introduced in the period from 2004 to 2014, by academia and industry sectors together. However, the currently available systems can be divided into two main kinds, graph databases, and graph processing. In this section and for the objectives of the present study, we will express the concepts of graph database systems in general, with a focus on Neo4j graph database system particularly in some cases. However, all these efforts in the field of graph modeling express the importance of graphs for real-world scenarios. Angles [20] summarized the advantages of using graphs as modeling mechanism for data management as following:

1. Graphs enable users to model data exactly as they are represented in the real-world scenario, this can significantly enhance the operations on data. Thus, graphs can keep all the information about an object in a single node and display the related information by relationships connected to it.
2. Queries can be developed based on the graph structure. For instance, the finding of the shortest path can be considered as subgraph from the original graph.
3. Operationally, graphs can be stored efficiently within databases using special graph storage structures, and functional graph algorithms for application of specific operations.

2.3.2 Graph database management systems

In general, a database is defined a set of related data that organized to represent a real-world situation in a way that supports manipulating of data and retrieving of information when required [20]. While, a graph database is considered as a special type of database, where graph components are used to display information about data interconnectivity or topology.

Nowadays, Neo4j graph database management system is the one of the simplest and powerful data management systems available for graph users. Neo4j [21] has been developed by Neo Technology to store data in form of an edge, a node or an attribute. Each node and edge can be labeled and could have a single attribute or any number of attributes. Nevertheless, Neo4j is provided in three different editions with different capabilities, which are Community edition, Enterprise edition and Government edition [21].

However, Neo4j graph database has been chosen to be used as a graph data management system for the current study due to the following reasons:

1. The Neo4j Community Edition is available for end-users as an open source graph database, licensed under the free GNU General Public License (GPL) [21], which enabling graph users to apply graph database to real-world problems.
2. Neo4j supports several modern query languages such as the RDF query language SPARQL, and the path-based query language Gremlin [8], in addition to its main expressive query language Cypher, which is considered the easiest graph query language.
3. Neo4j provides two mechanisms to import bulk data to develop graph model by transferring data from an external data legacy platform. This makes Neo4j the best graph database system for the present study.

The browser interface of Neo4j consists of several interactive windows as shown in Figure 2.20. Queries can be managed through the editor which is located along the top, where, the user can execute and run Cypher commands using Cypher query language of Neo4j. The commands outcomes are shown in the stream below the editor, this results frame allows the user to represent the results on one of the following forms:

1. Graphical representation (nodes and relationships), in case, that ‘Graph’ button is selected.
2. Textual representation, in case, that ‘Rows’ button is selected.

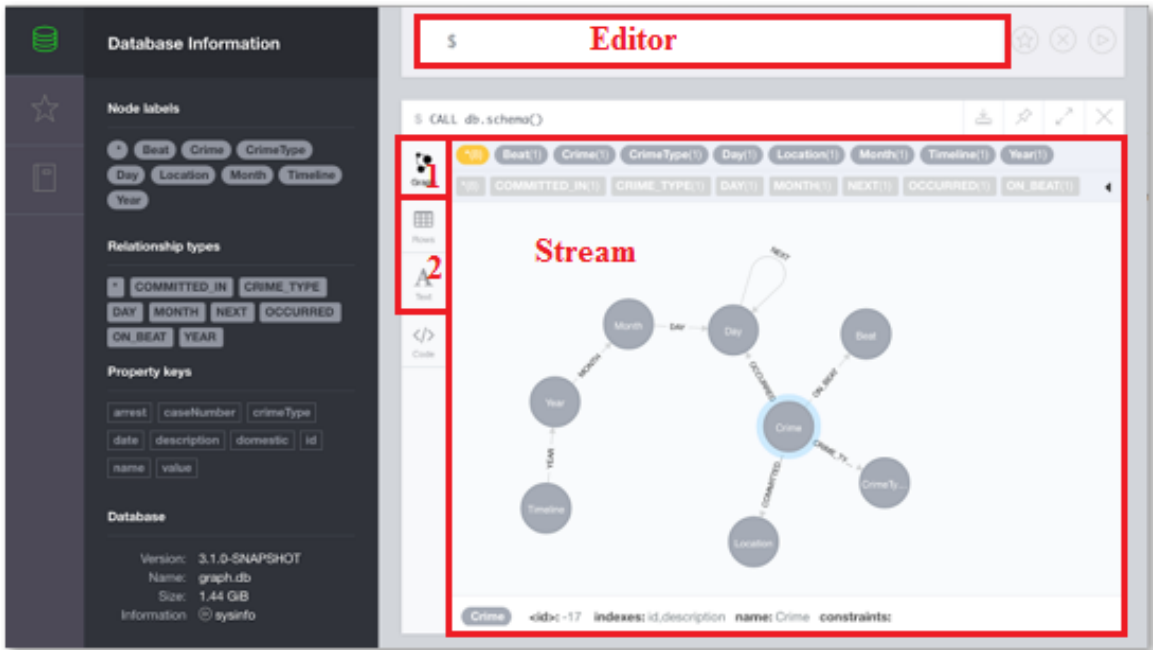

Figure 2.20: Neo4j browser interface

2.3.3 Graph database query languages

In general, query language can be defined as a set of operators or commands that can be executed to manipulated or query data within a database or information system. Recently, there are several computer query languages available to perform queries in databases. And yet compact, Cypher is one of these declarative graph query languages that enable users to programmatically represent graphs as diagrams using Neo4j graph database [17], and then apply filters and information retrieval queries. Cypher creates diagrams by following the same way that diagrams are been drawn on the whiteboard using ASCII art, as following:

Nodes

As an equivalent to ASCII, nodes are represented in Cypher by using parentheses, whilst, node labels are prefixed by a colon. Then, properties can be added to the node using curly braces, so that, the property key and the property value are specified within the curly braces.

```
(a: b {c: d})
```

where `a` node identifier
`b` node label
`c` property key
`d` property value

Relationships

Relationships are drawn using pairs of dashes combined with greater-than or less-than signs. The greater-than and less-than signs indicate the relationship direction. Square brackets are provided between the dashes to set down relationship name, and to specify the relationship property key-value pairs within curly braces inside the square brackets.

```
-[a: b {c: d}]-> or <-[a: b {c: d}]-
```

where `a` relationship identifier
`b` relationship label
`c` property key
`d` property value

2.3.4 Building a graph database model

Normally, the first stage in the modeling of graph database only includes some efforts to understand the entities of each domain, and how these entities are going to interconnect with each other. A miniature model for the entire informal model can be drawn on a whiteboard, where the most important nodes and relationships are shown; here each single node can represent a group of entities. Only then, the targeted graph model can be developed based on the general structure that acquired from the informal primary model. In Neo4j database the structure of any graph database is developed based on the relationship names and node labels. However, in most of the cases if not all, the process of building a database does not begin with a blank store followed by excessive use of CREATE statements to add data into the graph database [17]. Instead, master data can be developed by transferring data from a data legacy platform. Therefore, Neo4j provides two mechanisms to import bulk data.

The first mechanism is based on 'neo4j-import' tool. This tool provides high data transfer rates of around 1,000,000 records per second, this high-performance in transferring data is achieved by adding individual layers until the entire data that extracted from the Comma-Separated Values (CSV) files, are transferred and the data store is completely built. Data extracted from these files can be used to create nodes or relationships data in the import tool by applying the following command line of Listing 2.2 in Neo4j database:

```
neo4j-import  -- into target_directory \
--nodes first_file.csv  --nodes second_file.csv  --relationships _file.csv
```

Listing 2.2: Neo4j-import command

The above Cypher script notifies 'Neo4j-import' to develop database store files and to locate them in the 'target_directory'. The rest of the above Cypher script example will then be executed for each CSV file as following:

- The first CSV file contains CSV data to create a set of nodes. While, the first line of the file includes metadata for the names, labels, and properties of each node.
- The second set of nodes is created using the same method that has been used in the first CSV file.
- In the above example, the third CSV file is used to create relationships based on the CSV data. Whereas, each line in this file must contain start and end node, and may contain data for one or more relationship properties.

While, the second mechanism to load bulk data from external systems into a live Neo4j graph database, is based on using Cypher's LOAD CSV command. Which provides the ability to import million or so items, making it the most suitable method for regular batch updates from upstream systems. The following example illustrates how data can be loaded into a live Neo4j database using the cypher LOAD CSV command:

```
LOAD CSV WITH HEADERS FROM 'file:///file_name.csv' AS var
```

Listing 2.3: Cypher's LOAD CSV command

This Cypher script informs the database that the user wants to load a CSV data from specific file base on the Uniform Resource Identifier (URI) of the file. First, the sentence WITH HEADERS tells the database that the CSV file includes named headers in its first row. Then, the last part of the script 'AS var' assigns the input file to a specific variable, here the variable is 'var'.

In the next step, the user should inform the database, what to do with the recently loaded data, this is achieved by declaring Cypher commands that service the goal.

2.3.5 Querying philosophy of Cypher

Cypher has been developed as an expressive language to enable developers and database professionals to look for specific information and data within any dataset model to answer realistic queries. These queries are always initiated from one or more well-known starting points, which are recognized as *bound* nodes. This is followed by the information about the required data. However, the simplest queries within Cypher statement consist of MATCH clause followed by RETURN clause, or it could contain a combination of MATCH clause and WHERE clause with RETURN clause, here the properties lookup by MATCH clause are moved to WHERE clause. Practically, queries within Neo4 database are achieved by following two fundamental steps:

2.3.5.1 Querying patterns

There are two types of query patterns that can be applied based on the kind of required information. The first type of query patterns is known as simple queries. This type is when the user is seeking to find general information, in this case, the user could use MATCH clause or MATCH clause

with WHERE clause to describe the required information. For example, Figure 2.21 illustrates a simple duplex apartment. The ground floor plan consists of three spaces (Kitchen, Livingroom and Guest toilet), while, four spaces (Bedroom, Balcony, Bathroom, and Store) are located on the first floor.

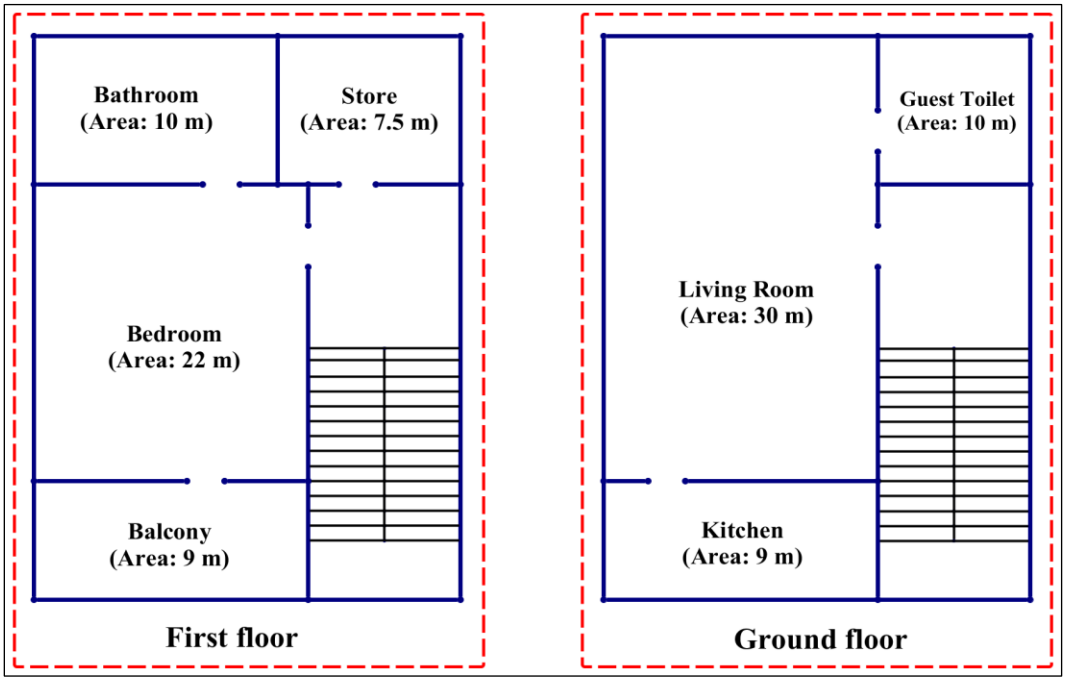

Figure 2.21: Duplex apartment floor plans

The duplex floor plans are represented by a simple diagram showing its spaces and floors in form of nodes, where the spaces are located. The simple diagram is shown in Figure 2.22.

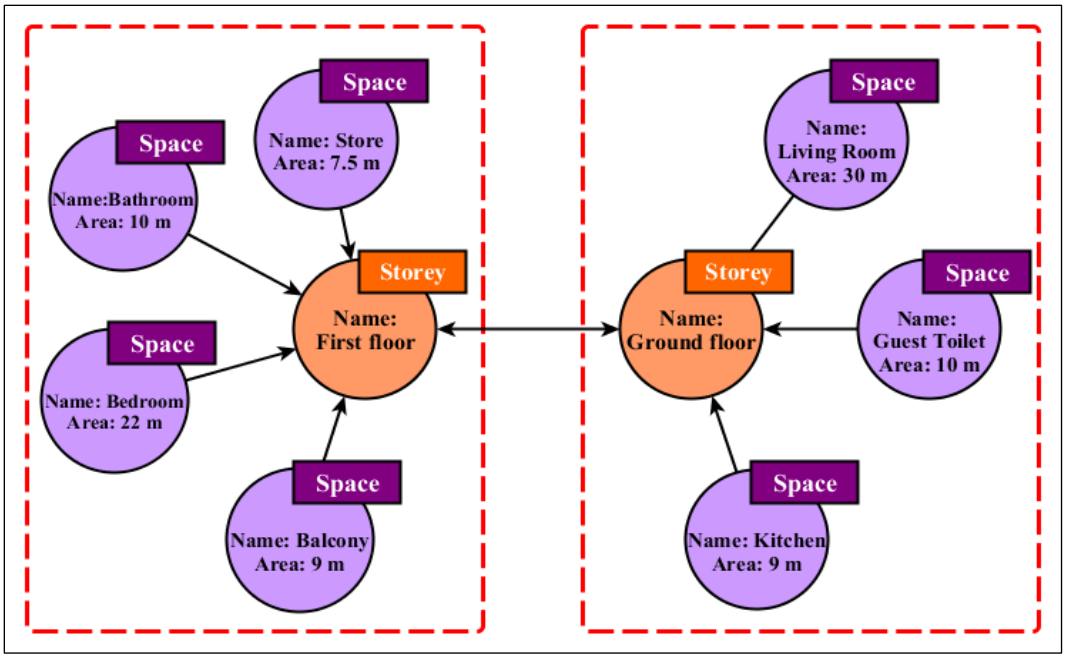

Figure 2.22: Duplex apartment plans graph

For instance, if the user is interested in reviewing all spaces in the duplex apartment, then the entity (Space) provides a starting point for the query. The search task can be achieved using Cypher command in Listing 2.4:

```
MATCH (s: Space) Return s as Spaces
```

Listing 2.4: Cypher command to return spaces in illustration example

Running Cypher query of Listing 2.4 returns all spaces that been stored in the graph database as shown in Table 2.1:

Table 2.1: Spaces within the duplex apartment model

Spaces
{name: Living_Room, area: 30}
{name: Kitchen, area: 9}
{name: Guest_Toilet, area: 10}
{name: Bedroom, area: 22}
{name: Balcony, area: 9}
{name: Store, area: 7.5}
{name: Bathroom, area: 10}

However, if the user is interested in reviewing all spaces that located within a certain floor, such as the first floor, then the entities (Space) and (Story) with name property value 'First_floor' provide the starting points for the query, as demonstrated in Listing 2.5 below:

```
MATCH (s: Space)-[r: located_in]->(storey: Storey {name: 'First_floor'})
RETURN s as Spaces_in_First_floor
```

Listing 2.5: Cypher command to return spaces located on a certain floor

The MATCH clause identifies all (Space) nodes and binds them to the identifier (a), while, the relationship will ensure that only (Space) nodes that located in the (c) should be returned. (c) is used as an identifier for (Storey) node with a property key (name) and property value (First_floor). Running this query returns the spaces within the first floor only, as shown in Table 2.2 below:

Table 2.2: Spaces within the first floor of the duplex apartment model

Spaces_in_First_floor
{name: Store, area: 7.5}
{name: Bathroom, area: 10}
{name: Bedroom, area: 22}
{name: Balcony, area: 9}

The second type of query patterns is known as complex queries. In this case, several syntactic elements are required in addition to MATCH clause to answer complex questions. Whereas, pattern nodes, arbitrary nodes, arbitrary depth paths and anonymous nodes are used -in addition to anchored nodes or starting nodes that discussed before- to support the query operation. Continuing with the simple duplex apartment model from the previous section, if one is seeking to retrieve all spaces with an area greater than 10 m, this can be achieved by running the following Cypher query in Listing 2.6:

```
MATCH (s: Space)
WHERE s.area > 10
RETURN s as Spaces_with_area_greater_than_10
```

Listing 2.6: Cypher command to return spaces with area greater than 10 m

Having applied the query of Listing 2.6 in the graph database, the following two spaces in Table 2.3 are acquired:

Table 2.3: Spaces with area greater than 10 m

Spaces_with_area_greater_than_10

{area: 30, name: Living_Room}
{area: 22, name: Bedroom}

2.3.5.2 Processing the results

In addition to declaring the information pattern that is desired, graph developers and users also should have to express the form of the outcomes. If the user is interested in a specific set of nodes, relationships or information, then there is need to constrain a set of results. Here, where the use of WHERE clause is raised, therefore the properties that identified by the MATCH clause are moved to the RETURN clause, to constrain the graph matches by eliminating the matched subgraph.

Chapter 3: IFC Meta Graph model

3.1 Introduction

This chapter presents a generic approach to create meta-graph models based on IFC EXPRESS schema, with the aim to graphically represent the IFC classes and the relationships among them, using Neo4j graph database management system. The IFC Meta Graph (IMG) models express the classes and the attributed that attached to them as nodes, and the relationships among them as edges, as shown in Figure 3.1. Several advantages can be acquired by having such graph models since it can be used as comparison tools to investigate updates and variations between different IFC EXPRESS schema versions. Furthermore, this meta-model can be used later as a reference model to generate additional relationships between IFC entities of the IFC Object Graph (IOG) models in the present work.

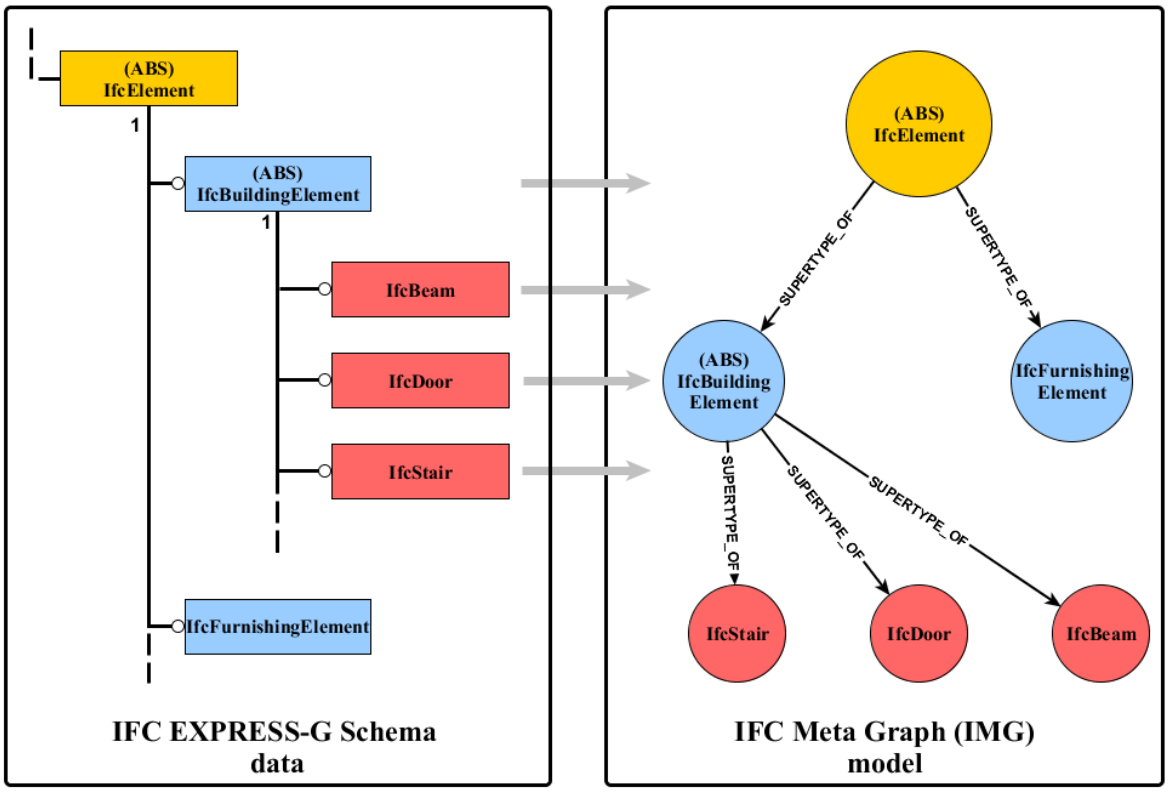

Figure 3.1: Graphical representation for IFCs using IFC Meta Graph model

buildingSMART [2] developed a graphical modeling using EXPRESS-G data modeling language to represent IFC classes, with aim of helping users in the understanding schema by representing a rectangular for all instances of the same class (objects) or data type. However, The IFC EXPRESS-G can provide a simplified display for the IFC data, but it does not represent all the IFC data, and it is just provided for reference purposes and not for implementation. Therefore, the IMG model

is introduced within the present study as an alternative tool to graphically represent the IFC schema. The mapping mechanism to generate the IMG model is presented in section (3.2) based on the IFC2x3 TC1 schema. Then, section 3.3 displays several graph query examples to evaluate the adequacy of the IMG model for analyzing IFC data and their relationships, and to provide suitable approaches for implementation of meta graphs in the real-world. This workflow is summarized below in Figure 3.2

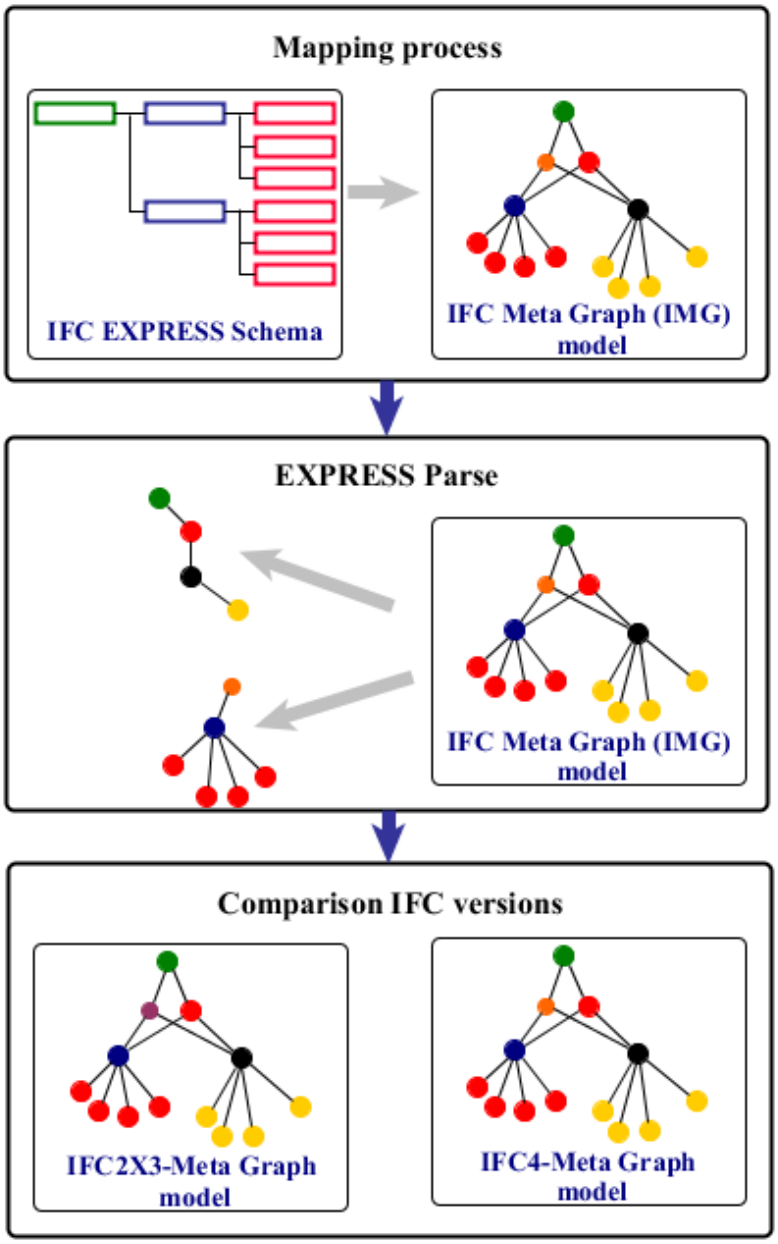

Figure 3.2: IFC Meta Graph (IMG) model workflow

3.2 Mapping mechanism

The IFC2X3 schema is used here as data legacy platform to build our meta-graph model. As depicted earlier in Chapter 2, IFC2X3 schema consists of 653 entities as it has 327 type definitions, in addition to 9 different domains [22], most of these data are represented within the IFC EXPRESS-G in a form of boxes and lines [23], as shown in Figure 3.3 below. Where, rectangular boxes are used to represent classes and data type, while, lines are used to describe the relationships among classes, class and its attributes, or supertype/subtype relationships.

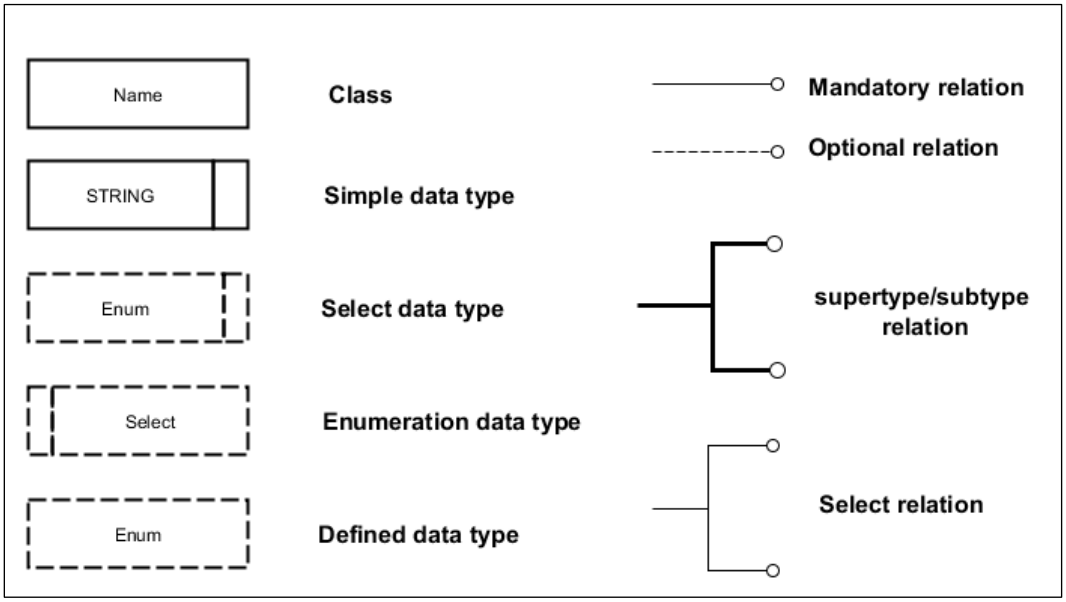

Figure 3.3: EXPRESS-G for geometric representation of objects and relationships

However, the IMG model in the present study describes the IFC data in a form of labeled nodes and directional edges, which is quite similar to the approach adopted by IFC EXPRESS-G. The major issue that should be addressed here is, which data will be presented as nodes and which data will be expressed as relationships. Therefore, the following sections explore the mapping mechanism in detail.

3.2.1 Classes and attributes

Firstly, all instances within the IFC database which are known as classes will be identified as nodes in the meta-graph. Each node is labeled with the class name that it represents, thus each node is describing one instance of a class (e.g. *IfcSlab*), as shown in Figure 3.4. Normally, the abstract supertype classes do not exist in themselves within the IFC data models, but their subtypes are presented. However, nodes within our IMG model will be built to represent abstract and non-abstract classes based on their original hierarchical order as it has been shown in Figure 3.1 of section 3.1.

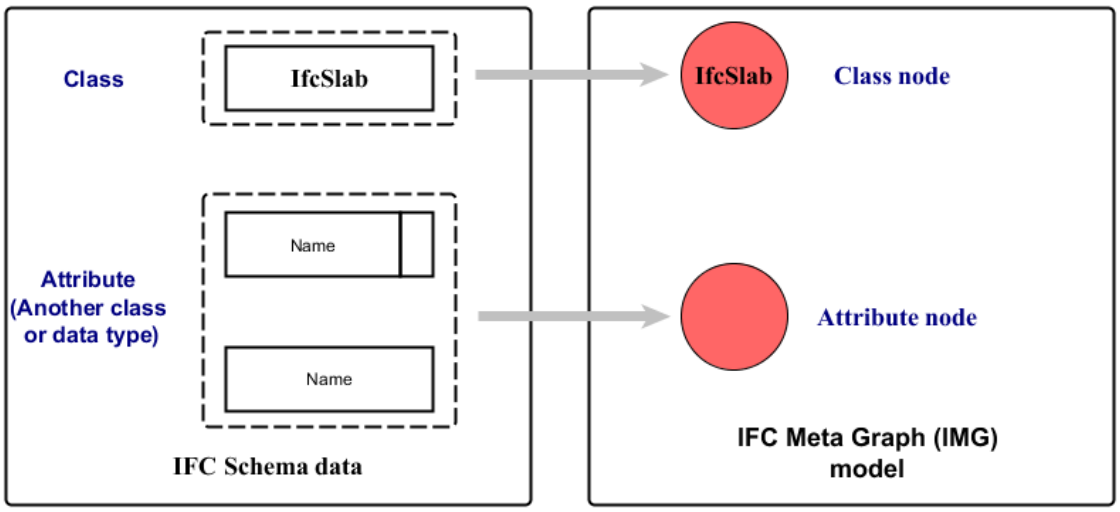

Figure 3.4: Mapping of classes and data types within IMG model

Secondly, data members such as simple data types or any type of attributes that listed earlier in Figure 3.3 of the previous section, which are normally attached to objects as attributes. These data members are going to be described as a node as well within the meta-graph model, as it is shown in Figure 3.4 above.

3.2.2 Relationships

There are several types of relationships defined by the IFC schema as it has been illustrated in Chapter 2. One of these relationships is the relationship which connects classes with their direct attached attribute. Here, it does not matter whether the attribute is just a data member or even another class, or whether the relationship is mandatory or optional, in any of the cases above the meta-graph represents these relationships as an edge line connecting two nodes. In fact, the edge line of the Meta graph model is a generic relationship qualified with name ‘has_property’. The ‘has_property’ relationship connects each class to it's (attribute definitions), which is represented as a node as well. Properties will be added to the target attribute definitions to indicate whether it is an optional or mandatory relation, as illustrated in Figure 3.5.

Relations can also exist between classes in a form of supertype or subtype for another class or a group of other classes. The supertype/subtype relation will be represented by the meta-graph model using the exact generic relationship that has been used earlier, but here, the relationship type is always fixed and called ‘SUBTYPE_OF’, as illustrated in Figure 3.5.

In the following example, the two type of relationships used in our IMG model, are explained using the *IfcBuildingStorey* class. This example is presented here to show how generic relationships are created using multiple Cypher commands similar to Cypher command in Listing 3.1 below:

```
MATCH (storey: IfcBuildingStorey),(element: IfcSpatialStructureElement)
CREATE (storey)-[r: SUBTYPE_OF]->(element)
MERGE (storey)-[:has_property]-> (n: Elevation{class: "IfcLengthMeasure",
optional: true})
```

Listing 3.1: Cypher command to create generic relationship in the IFC Meta Graph model

The Cypher command in Listing 3.1 will create relationships with the following characteristics:

1. The CREATE command create an (SUBTYPE_OF) relationship between the *IfcBuildingStorey* class and its first level supertype class, which is *IfcSpatialStructure-Element*.
2. While, the MERGE command create a (has_property) relationship between the *IfcBuildingStorey* class and its attribute definition (Elevation), which has a property pair {Optional: true} to indicate if it is a mandatory or optional relationship.

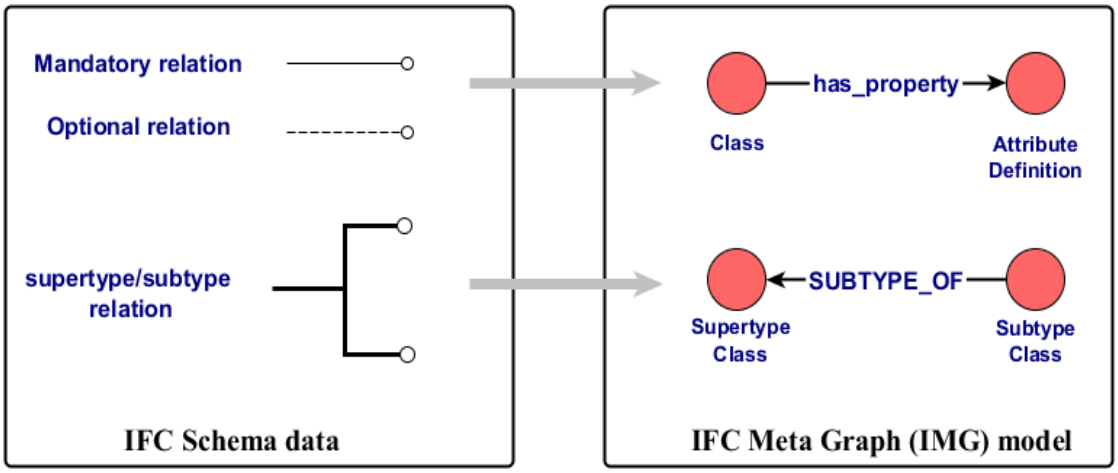

Figure 3.5: Mapping of relationships within IMG model

The entire multi-statement Cypher script to create the IMG model for the IFC2X3 schema in this study is shown in Appendix A.2. This script has been generated by applying Ruby [24] script that shown in the Appendix A.1. This script has been applied in the Domain Specific Language (DSL) interface of the online IFC data model server IFCWebServer.org [25].

3.3 Validation and analyzing of IMG

Having established our IMG model using the mapping mechanism that proposed earlier, where, IFC data have been successfully stored within Neo4j graph database in form of labeled nodes and generic relationships. It is time for querying and analyzing the generated meta-graph model.

3.3.1 Data profiling

The data profiling is performed to examine the correctness of the IMG model that generated earlier using Neo4j. Data retrieved from the meta-graph database will be matched with information extracted from the IFC2X3 schema. To ensure that our meta-graph model comprises of 653 entities and 327 type definitions and relationships, as presented by the IFC2X3 schema. Firstly, the total number of classes that have been stored as nodes within the meta-graph model will be specified by applying the following Cypher query scripts in Listing 3.2:

```
MATCH (n {Version: 'IFC2x3'})
RETURN COUNT (n) AS Total_number_of_IFC2x3_Schema_entities
```

Listing 3.2: Cypher command to count total number of IFC2X3 entities

The graph database has shown that there are 653 entities with the IFC2X3 meta-graph model as shown in the snapshot of Neo4j's interface stream (Figure 3.6).

Figure 3.6: Total number of IFC2X3 entities as exist in Neo4j (snapshot)

Up to this level of data profiling, the IFC classes and their relation with the correspondent parent classes are examined. This data filtering is conducted to ensure that each class has the correct relationship with its supertype classes (parent class) and whether the (SUBTYPE_OF) relationships that created earlier are correctly represented. The following Cypher query script in Listing 3.3 can easily specify each class and return a list contains all classes and their parent class.

```
MATCH (a {Version: 'IFC2x3'})-[r: SUBTYPE_OF]->(b)
WITH DISTINCT LABELS (a) AS LABELS1, LABELS (b) AS LABELS2
UNWIND LABELS1 AS Class_name
UNWIND LABELS2 AS Parent_class
RETURN Class_name, Parent_class
ORDER BY Class_name
```

Listing 3.3: Cypher command to acquire list of IFCs and their supertype classes

Having applying Cypher command in Listing 3.3, a list contains all the IFC classes and their parent classes, can be retrieved. The entire list is shown in Appendix A.1, however, an excerpt of that list is shown here in Table 3.1 below.

Table 3.1: Excerpt from finding list of IFC2X3 classes and their parent classes

No.	Class_name	Parent_class
036	IfcBeam	IfcBuildingElement
037	IfcBeamType	IfcBuildingElementType
038	IfcBezierCurve	IfcBSplineCurve
039	IfcBlobTexture	IfcSurfaceTexture
040	IfcBlock	IfcCsgPrimitive3D
041	IfcBoilerType	IfcEnergyConversionDeviceType
042	IfcBooleanClippingResult	IfcBooleanResult
043	IfcBooleanResult	IfcGeometricRepresentationItem

The complete version of the above table can be used to verify the hierarchical order of any IFCs and their relationship status, by matching classes and relationships.

3.3.2 Analysis of hierarchical structure of IFC classes

The IFC schema file is hierarchical database contains several types of relationships as it has been introduced. Therefore, it is valuable to generate a queryable tool for users to run complex queries and acquire information which supports understanding of the IFC schema. For instance, if a user is interested to know how deep is a certain class or an object from another supertype class, or how many intermediate classes are there along the path, then the meta-graph model can be a great tool for such type of queries. Let us take the *IfcSlab* class as an example, by running Cypher query in Listing 3.4 we will be able to know, how many time the *IfcSlab* is subtyped from the supertype class *IfcRoot* in the IFC2X3 schema:

```
MATCH (root: IfcRoot {Version: 'IFC2x3'})<-[r*]-(slab: IfcSlab)
WITH slab, LENGTH(r) AS is_sybtyped_times_from
RETURN LABELS(slab) AS The_class, is_sybtyped_times_from, LABELS(root) AS
The_supertype_class
```

Listing 3.4: Cypher command to check deep of a certain class from another supertype class

The query outcome had shown that the *IfcSlab* is subtyped six times from the supertype class *IfcRoot* as it shown in the snapshot of Figure 3.7 below:

The_class	is_sybtyped_times_from	The_supertype_class
[IfcSlab]	6	[IfcRoot]

Figure 3.7: Finding deep of a certain class from a supertype class

In the next example, the Cypher query in Listing 3.5, is implemented to show the entire path from *IfcRoot* class down to *IfcSlab* class, and the intermediate classes in the same time:

```
MATCH (root: IfcRoot {Version: 'IFC2x3'})-[r*]-(slab: IfcSlab)
WITH slab, r AS Show_path_from_root
RETURN Show_path_from_root
```

Listing 3.5: Cypher command to show the path between two classes

The graph database has shown the following relationships path between *IfcRoot* class and *IfcSlab* class as displayed in snapshot of the below figure (Figure 3.8)

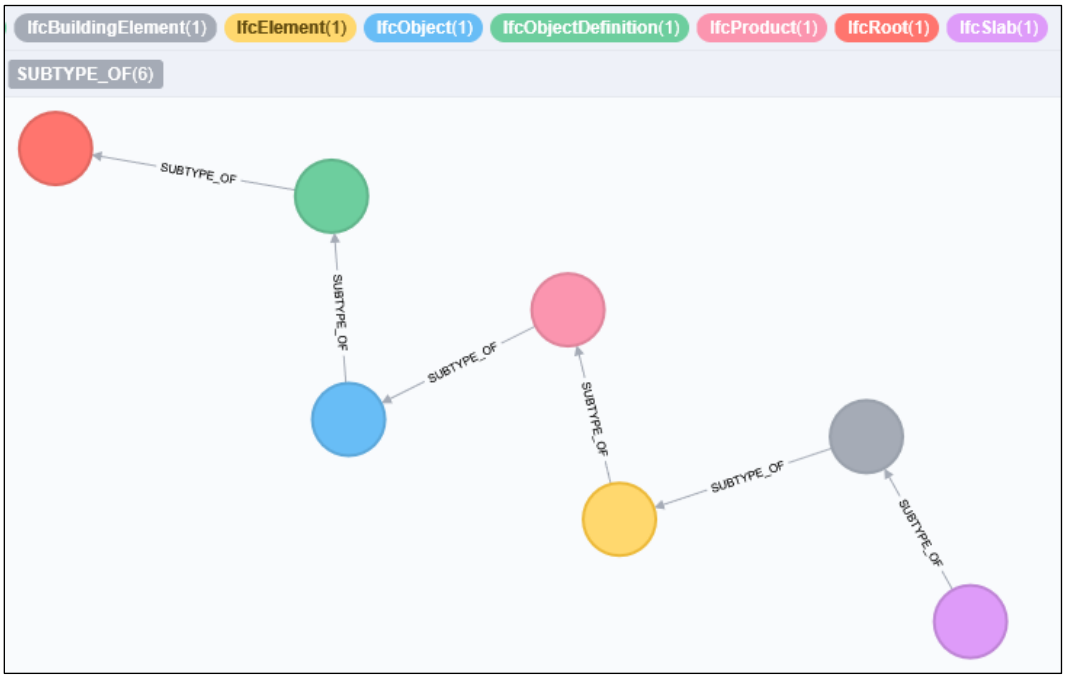

Figure 3.8: Relationships path between *IfcRoot* class and *IfcSlab* class

The above graph representation can clearly display the type *IfcSlab* is subtyped six times from the root of the entity hierarchy.

The final example in this section displays one of the most frequent forms of requested information by IFC schema users, which is super abstract classes of a certain class. Through the series of subtype relations that defined earlier as a path within the IMG model. The user can easily acquire the head of a path or all the supertype classes from which the class understudy is subtyped. Continuing with the previous example of the slab the following Cypher graph query in Listing 3.6. can return a list of all supertype classes to which the slab belonged:

```
MATCH (n)<-[:SUBTYPE_OF*]-(b: IfcSlab {Version: 'IFC2x3'})
OPTIONAL MATCH ()<-[r:SUBTYPE_OF]-(n)
WHERE r = null
RETURN LABELS(n) AS Supertype classes
```

Listing 3.6: Cypher command to acquire the series of supertype classes to *IfcSlab* class

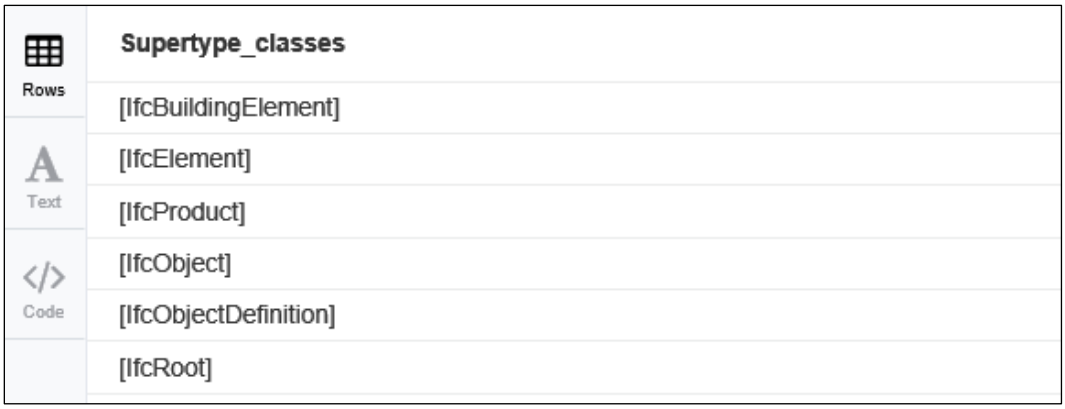

Figure 3.9: The acquired series of supertype classes to *IfcSlab* class

The outcome of the last query is shown in Figure 3.9 above. The list of the retrieved classes matches the classes that had been shown within the relationships path in Figure 3.8.

3.4 Comparison of different IFC schemas

The present section proposes an approach based on graph query to compare different IFC standard versions released at the different period to trace the new features. Firstly, the IFC standard versions will be stored in a single graph database in form of IMG models. The IMG models will be designed by considering the queries that will be run on the graph. Therefore, each IMG model will be created with unique property values to ensure uniqueness. To indicate the features that have been added or modified in any IFC schema version in compare to its previous version, and vice versa, the following query command in Listing 3.7 can be used.

```
MATCH (a {MetaModel: 'IFC2x3'}), (b {MetaModel: 'IFC4'})
WITH [a] as New_Standard, [b] as Existing_Standard
RETURN FILTER (n IN New_Standard WHERE NOT n IN Existing_ Standard) as
New_Items
```

Listing 3.7: Cypher command to trace new items in updated version model

One can trace the history of IFC standardization process by conducting a series of graph queries based on the comparison idea that presented earlier. Since the comparison of different IFC versions is not the main objective of the present study, therefore, the last query will not be applied here.

3.5 Discussion

The IMG model can be considered as referenced graph database for meta information of IFC EXPRESS data model. This reference database will be used later to simplify the data retrieval from IOG databases.

The advantages of utilizing IMG model can be summarized as following:

1. The IMG provides a textual and graphical representation of IFC schema data, using a single graph database management system. While two different systems are required in case of using IFC specification as standard data to describe the IFC schema data, the first system is for textual representation which is provided by IFC EXPRESS specification documentation, and the second system is the graphical representation which is provided by EXPRESS-G.
2. The IMG model can be considered as an interactive database system, where the user can run filters and information retrieval queries for a better understanding of IFC schema, and analyzing its complex data connectivity.
3. The IMG model provides a platform for comparing two or more IFC versions which are stored in Neo4j as graph-based models, where each node could have a special attribute to identify to which IFC versions it belongs.

Despite the querying capabilities that provided by the IMG model in this chapter, it is still cannot fully represent all IFC data within the IFC schema, such as inverse relationships and page reference.

Chapter 4: IFC Object Graph model

4.1 Introduction

The present chapter demonstrates the methodology that been developed to create IFC Object Graph (IOG) models based on IFC data extracted from IFC object models. Figure 4.1 shows the process of developing IOG models database comprise of two main stages. In the first stage, the IFC data are extracted from the IFC model SPF file and converted into tabular data in a form of Comma Separated Values (CSV), then these data are imported into Neo4j graph database to create the IOG model. While the second stage involves data enhancement processes to enable users to storage multiple IOG models in a single graph database, in addition to the use of multi-labels nodes to support the implementation of queries and filters in a later stage.

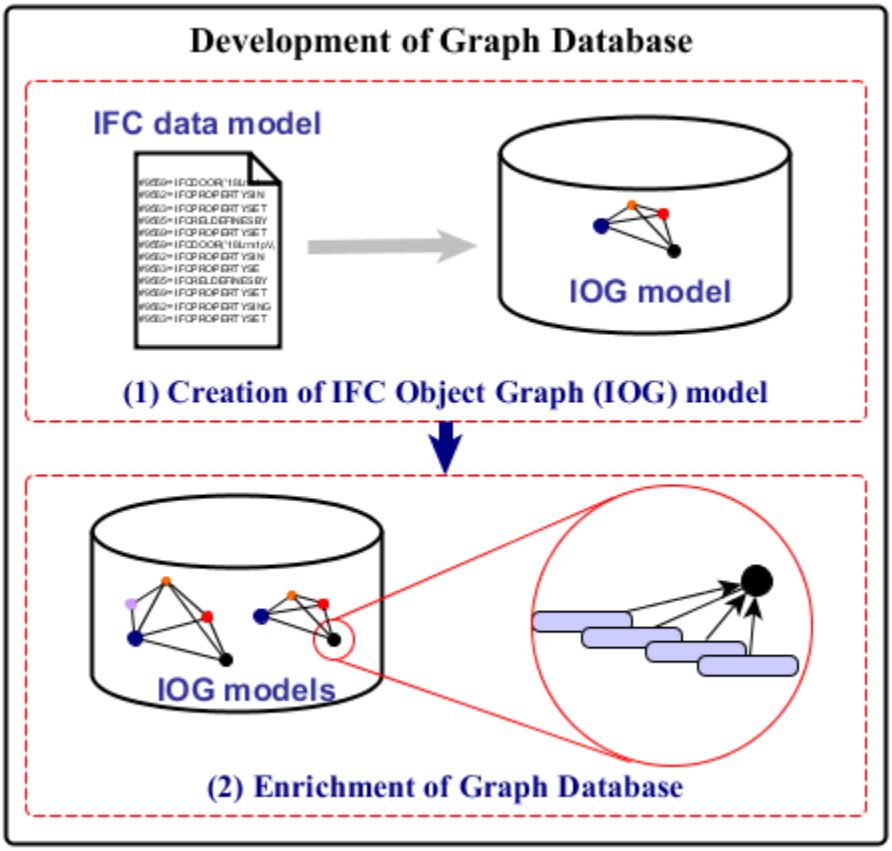

Figure 4.1: Processes of developing IFC Object Graph (IOG) models database

4.2 Building-up IFC Object Graph model

This section introduces two approaches to generate and develop IOG models, the both approaches are using the same technique to transfer the IFC data into the Neo4j graph database. However, the first approach is based on hand coding and manual execution of Cypher commands, while the

second approach makes use of programming languages and web applications to automate the processes of graph generation.

The key idea is that the IFC data cannot be imported directly into Neo4j graph database from the IFC file in STEP physical file format. Therefore, the adopted technique proposes conversion of IFC data into CSV format, and then import them into Neo4j to create the graph-based models. The proposed technique for graph modeling, consists of two steps; (1) IFC data preparation and (2) graph generation (Figure 4.2).

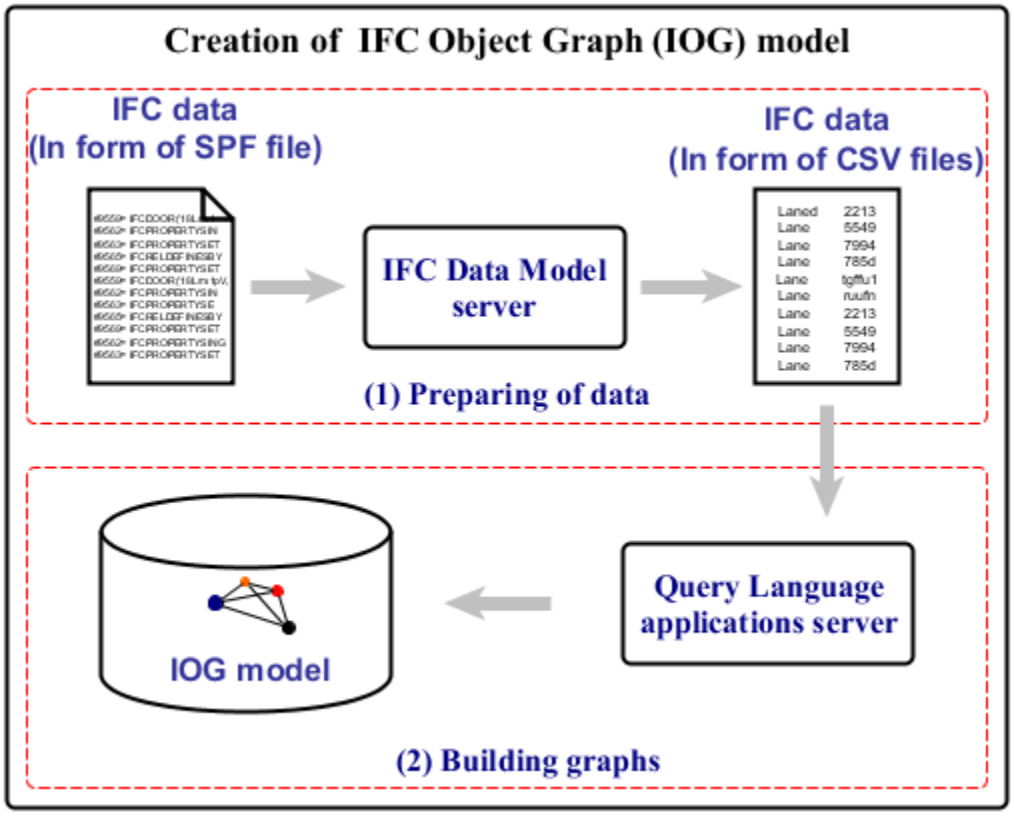

Figure 4.2: Processes of building-up IFC Object Graph (IOG) models

4.2.1 Manual generation of IOG models

The present section describes the manual processes of IFC Object graph models formation, where the two main steps of this operation are presented in detail. The formation operation has been executed manually using a simple IFC model, firstly. Wherefore, it should be mentioned that this approach could not be a practical approach when working with large or disparate datasets, however, it is introduced here in advance to the automatic approach, to create a solid understanding of the data transmission process, since that several details could remain unseen in case of automatic generating for graph.

4.2.1.1 IFC data into CSV data

As an initial step, the IFC data will be reorganized in a form of tabular data, particularly as CSV format, only then, it can be imported into the Neo4j graph database. Since that Neo4j imports bulk data that stored in a tabular data format only, using the LOAD CSV command which has been introduced earlier in chapter 2. In fact, the Comma Separated Values (CSV) file format has been used widely as data exchange tool for several spreadsheet programs. where data can be encoded in lines for storage, each line consists of one or several fields that are divided by commas [26]. Furthermore, the CSV file could contain a header line at top of the file, where the single name can be placed in each field of the header line. Therefore, each column of data sets is represented vertically by one name from the header row.

For storing IFC data in CSV files, the data fields of the header line are used to represent the name of the attributes. Therefore, each column of attribute values is represented vertically by a single attribute name from the header row, since each CSV file represents only single IFC class. Figure 4.4 illustrates the conversion process of the IFC data in STEP format into CSV format while showing an example of CSV file to represent a class called *IfcDoor*. The class attributes (IFCID, Name and OverallHeight) are represented in the header line, while the attribute values are shown as three different columns of datasets.

Several CSV files should be created for each IFC model based on the number of the classes. This is achieved by using the IFCWebServer.org [25]. The IFCWebServer provides a simple mechanism to extract and export user-defined reports, where the user can easily export data in different formats (HTML, XML, CSV) and simply define the information that should be presented in the report. Such method could be a time-consuming process because each IFC class-type entities should be generated separately. However, on the other hand, this approach makes the user more flexible to choose which data should be reported, and which data should not be presented.

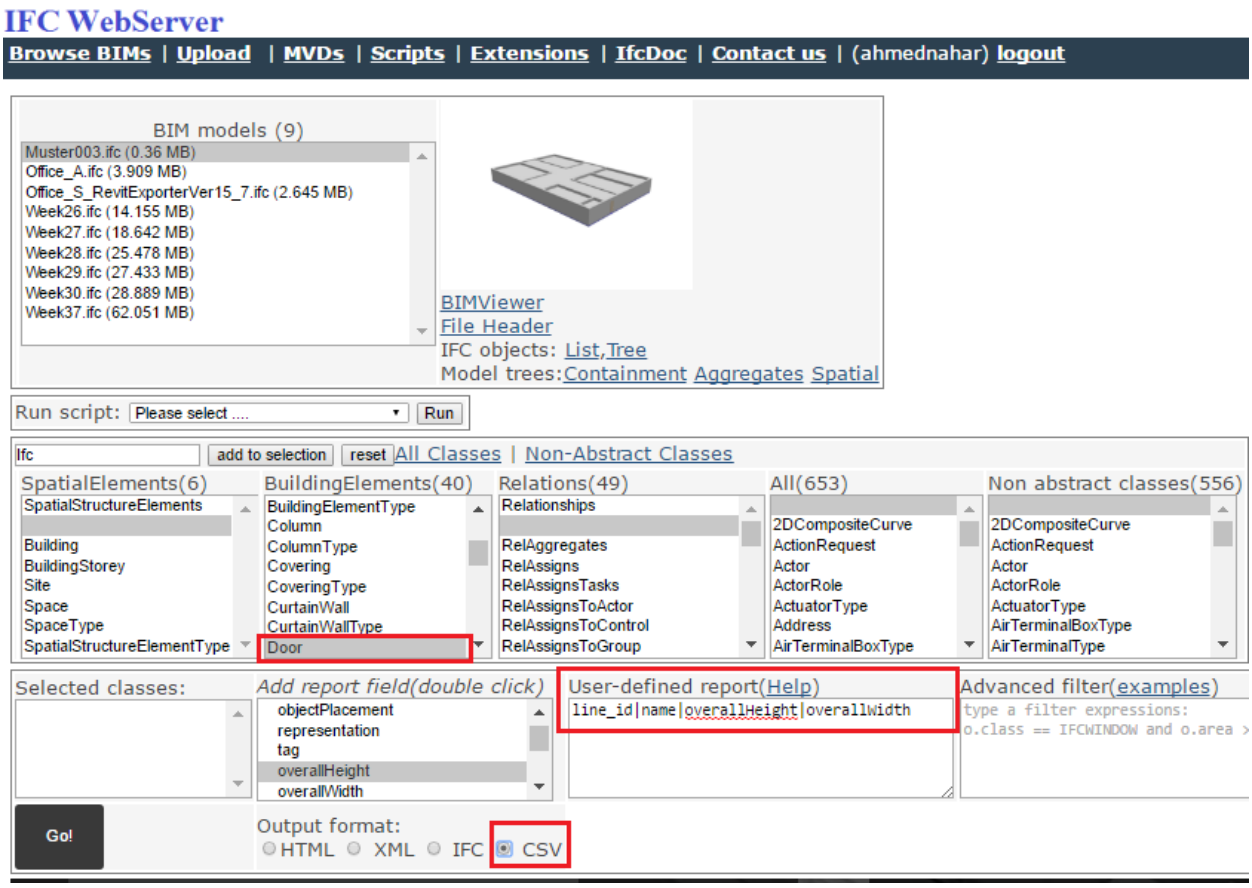

Figure 4.3: Running a user-defined report query in IFCWebServer.org

Continuing with the example of the *IfcDoor* class, and after uploading the IFC model file into the IFCWebServer, the user can easily define the three attributes that should be reported by simply typing the filter expression in the 'user-defined report' interface, and then export the required data in form of CSV file (Figure 4.3), where, the number of file lines depends on the number of door instances of same class (objects).

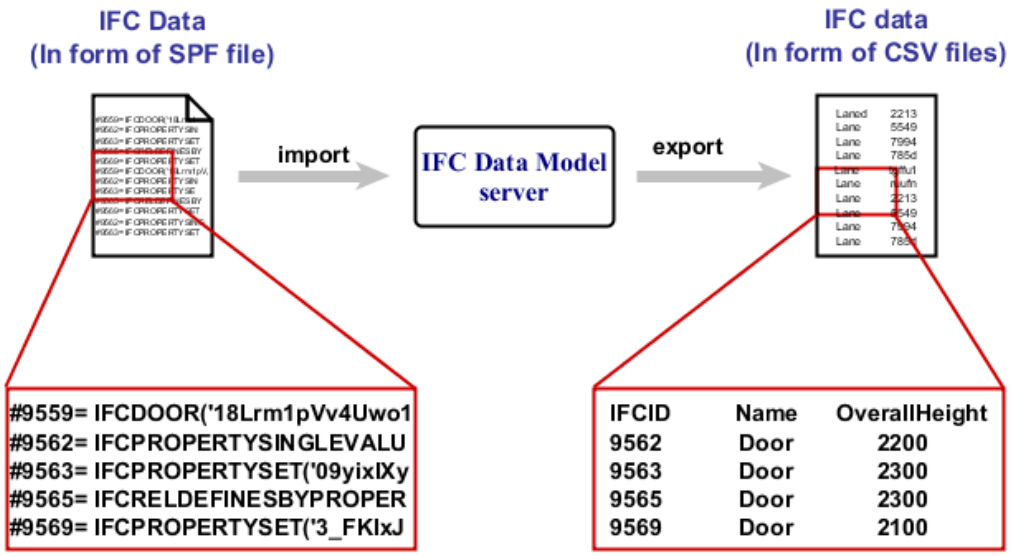

Figure 4.4: Conversion of extracted IFC data into CSV format

4.2.1.2 Import CSV data to graph database

For developing IOG models using Neo4j, the nodes and the relationships will be created based on data extracted from the CSV files that generated in the previous step. Thus, there is a need for a tool or a mechanism to migrate these data from the CSV files into a live Neo4j database. A widely-used approach in the case of importing a bulk data into a live Neo4j database is Cypher LOAD CSV command that introduced in chapter 2. This command will be used several times to create nodes and relationships, as depicted in Figure 4.5. Therefore, the CSV files are used two times, firstly, to create the nodes which represent the IFC entities and secondly, to create the relationships among these entities.

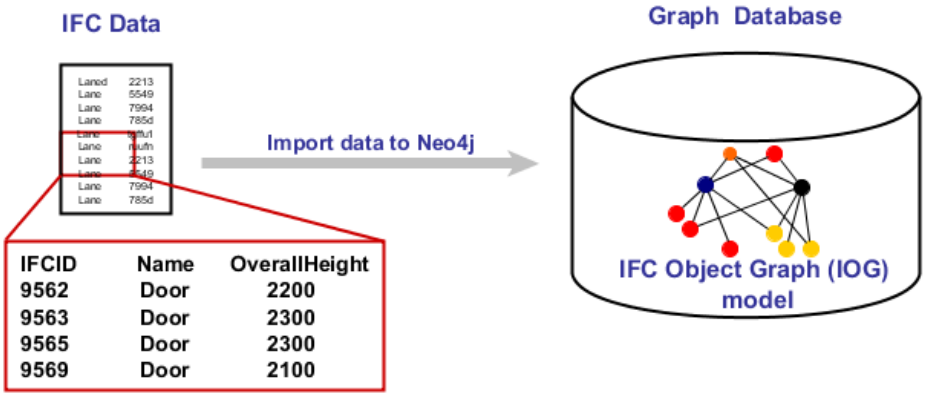

Figure 4.5: Importing IFC data into graph database

Representation of entities

A simple technique has been used to create the nodes and their properties which represent the IFC entities, using data extracted from CSV files. As mentioned earlier, each header line contains metadata of each class which are the property keys of this IFC class. As an example, if the user interested to create nodes that represent *IfcMaterial* entities using data from the CSV file as shown in Listing 4.1 below. Here, the *IfcMaterial* class has only two attributes which are; IFCID and name. Moreover, there are five material instances of same *IfcMaterial* class.

```
1     IFCID       name
2     9925        In-situ_concrete
3     9472        Metal
4     9471        Wood
5     1456        Masonry
6     1433        Thermal_insulation
```

Listing 4.1: Example of IFC data within a CSV file

To transfer all these data to Neo4j database and create five nodes, where each node holds the same label *IfcMaterial* and two different property (IFCID and name). Then, the following Cypher command of Listing 4.2 is used:

```
LOAD CSV WITH HEADERS FROM "file:///Muster003/IfcMaterial.csv" AS line
FIELDTERMINATOR' '
CREATE (material: IfcMaterial {IFCID: line.IFCID, name: line.name})
```

Listing 4.2: Cypher command to create node using data from CSV file

The Cypher command of Listing 4.2 informs the database that the user wants to load CSV data from the URI "file:///Muster003/IfcMaterial.csv", and create nodes with label *IfcMaterial* and two attribute keys, and get the attribute values for each attribute key from the CSV file.

Representation of relationships

Up to this level, the graph database consists of only the IFC entities as graph's nodes accompanied with the following characteristics:

1. Node's labels representing the IFC class-names.
2. Node's properties, which are the attributes.

In the next level, the fragmented nodes stored in the Neo4j database will be connected to each other, using Cypher commands for relationships generation. Therefore, the mechanism of creating relationships highly depends on nodes' properties. Since that, each IFC entity has a unique STEP-Id which is represented along this study as IFCID. This unique IFCID is used to trace back the

correct start node of the relationship within the graph database, and then to be linked with end node based on it is IFCID as well. However, each relationship holds a name which is presented within the graph database as relationship type. The relationship name is acquired from the attribute names that presented in the header line of each CSV file if the attribute defines another class and not just a data member.

As an illustration example, if the user wants to create a *representedMaterial* relationship to link each *IfcMaterialDefinitionRepresentation* class entity with its *IfcMaterial* entity. To achieve this goal, both group of entities should be represented first as nodes within the graph database by applying the procedure presented earlier in a section of nodes representation using the following CSV files in Listing 4.3 below. The first file represents the *IfcMaterial* nodes and their attributes, while the second file represents the *IfcMaterialDefinitionRepresentation entities* with attribute *representedMaterial*, this attribute is used to create a directional relationship between the nodes of the second file to nodes in the first file by matching their unique IFCID each time.

```
1     IFCID       name
2     9925        In-situ_concrete
3     9472        Metal
4     9471        Wood
5     1456        Masonry
6     1433        Thermal_insulation
```

(a)

```
1     IFCID       representedMaterial
2     9944        9925
3     9945        9472
4     1489        9471
5     1487        1456
6     1451        1433
```

(b)

Listing 4.3: Excerpt (a) *IfcMaterial* and (b) *IfcMaterialDefinitionRepresentation* CSV files

To transfer these data to Neo4j database and create five relationships, here each relationship will hold the name *representedMaterial*. The following Cypher command of Listing 4.4 below, can be used to achieve this goal:

```
LOAD CSV WITH HEADERS FROM
"file:///Muster003/IfcMaterialDefinitionRepresentation.csv" AS line
FIELDTERMINATOR' '
MATCH (mDef: IfcMaterialDefinitionRepresentation {IFCID: line.IFCID}),(
material: IfcMaterial{IFCID: line.representedMaterial})
CREATE (mDef)-[r:representedMaterial]->(material);
```

Listing 4.4: Cypher command to create node using data from CSV file

Cypher command of Listing 4.4 loads CSV data from the URI "file:///Muster003/IfcMaterialDefinitionRepresentation.csv". and matches the IFCID of each *IfcMaterialDefinitionRepresentation* node with IFCID loaded from the CSV file, then links the successful match with IFCID of the *IfcMaterial* node based on the *representedMaterial* dataset column of the CSV file. Finally, create relationships hold the name *representedMaterial* for each successful match from above.

4.2.2 Automatic generation of IOG models

The manual approach that introduced in section 4.2.1 above, to develop graph database, could not be a practical approach when working with large or disparate datasets. But, it can be considered as an appropriate approach for understanding the data transmission processes and the concept of building IOG models in detail. Therefore, to provide a practical approach for IFC data transmission process within this study, a proposal has been put forward to make use of the available online data model servers for BIM, which allow users to automatically export the data inside IFC file into different formats such as spreadsheets. The present approach to automating the data transmission processes consists of two steps. These two steps are used to overcome the manual repetitive subtasks of the hand coding method, and to make use of the programming languages and web applications as following:

4.2.2.1 Extraction of IFC data

Here, all the CSV files that contain the IFC data will be generated in a single step, in addition to other two text files. The first text file contains all the LOAD CSV commands which will be used later to load and generate the labeled nodes based on data extracted from the CSV files. While the second text file contains a series of Cypher commands to generate relationships among the nodes stored in Neo4j graph database. In this step, the CSV files plus the two Cypher scripts files are automatically generated without accessing the original IFC file, simply by using the domain specific language (DSL) of the online data model server that introduced earlier. The IFCWebServer provides Domain Specific Language (DSL) for users with Ruby [24] language programming skills.

The Ruby script that been used here to generate the CSV files and the multi-commands text files for first IFC model within the present study, is shown in the Appendix A.3. This script has been executed in the IFCWebServer to acquire the following outcomes:

1. CSV files, each file contains data the correspond to a single IFC class-type. The IFC class-type attributes are presented in the file header, while the remaining rows contain the IFC class-type entities.
2. Text file contains all the Cypher commands to import IFC data from each CSV file and create the corresponding nodes (The multi-commands Cypher file to generate graph entities for the First IFC model in this study is shown in the Appendix A.4)

3. Text file contains all the Cypher commands to create relationships among the nodes exist in the graph database.

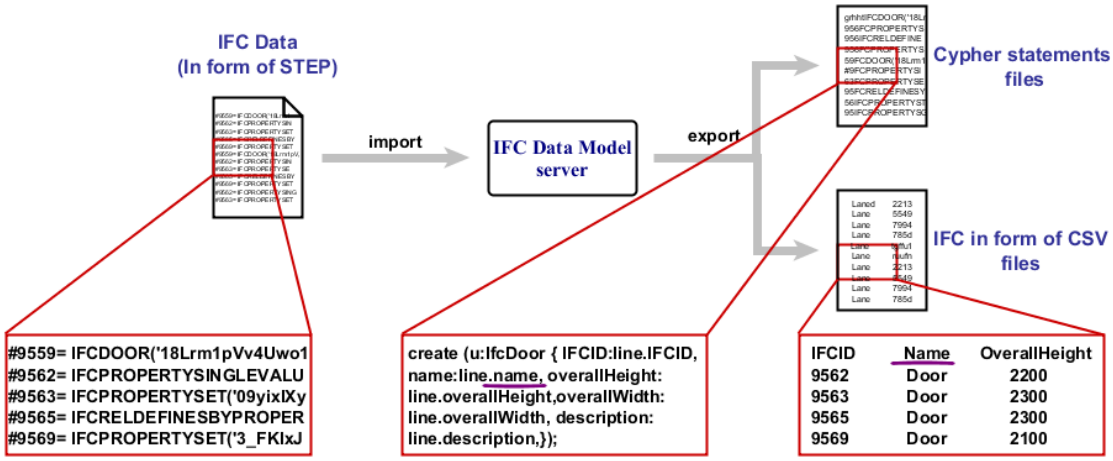

Figure 4.6: Automatic generation of the CSV files plus the multi-commands Cypher files

4.2.2.2 Execution of Cypher scripts

The multi-statement Cypher script that automatically generated in the previous step, and shown in Appendix A.4, has been executed one at a time using the web application 'LazyWebCypher' to generate the graph model. The LazyWebCypher [27] allows the users to simply load Cypher script that contains multiple transactions, which could be a satisfactory tool in many cases such as our study, where several Cypher commands such as the LOAD CSV commands, can be executed once. Thus, the LOAD CSV commands in addition to Cypher commands to create relationships among the IFC objects, are supplied to LazyWebCypher at one time. Then, as soon as the Cypher scripts are run and executed, the entire IFC data will be directly imported to the live Neo4j instance.

4.3 Enhancement of Graph database

In this section, two enhancement processes are introduced to improve the querying and filtering quality of the graph database in this study by making use of graph capabilities. Firstly, the single graph database is used to store several IOG models. In addition to the enhancement of the IOG models themselves where each node can hold multiple labels. The proposed enhancement processes are demonstrated in Figure 4.7 below and explained in detail in the next following sections.

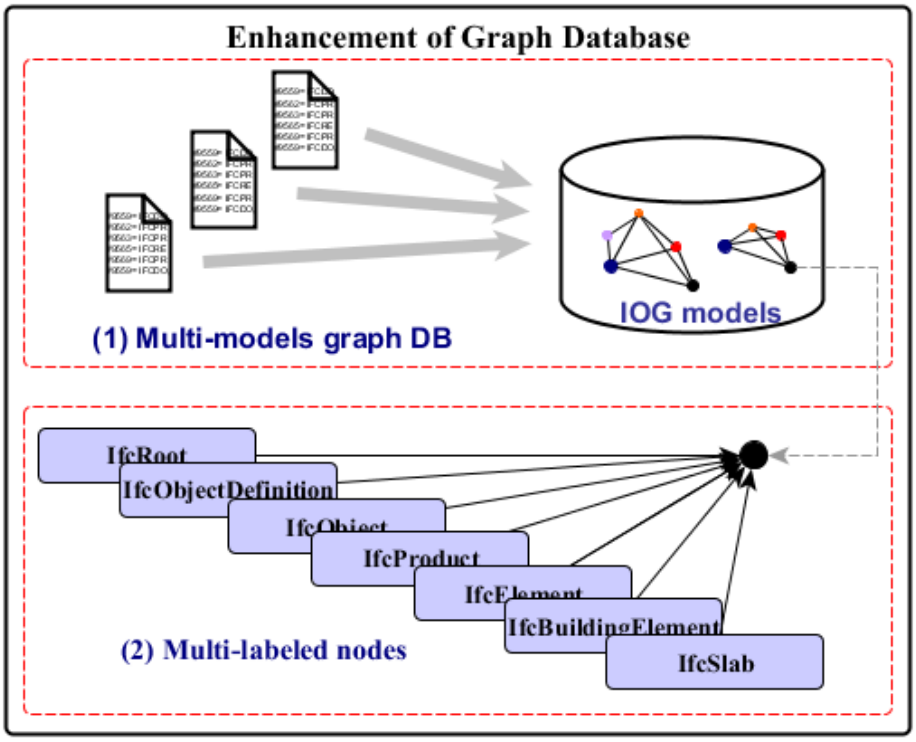

Figure 4.7: Enhancement of the graph database

4.3.1 Multi-models DB

The first approach for graph database enhancement by having multiple IFC models stored within a single graph database is demonstrated. This objective is achieved by making use of labeled property graphs that introduced earlier in chapter 2. So, that any group of nodes can easily be distinguished from the remaining nodes through their unique properties even if they have the same label. Making use of these characteristics, users are apple to storage data from different model sources in the same Neo4j database.

Graph modeling of several IFC models

The IFC models that shown earlier in the previous section will be stored in a Neo4j database using a simple technique to allow the graph database to differentiate between data from different sources. In fact, data from different IFC models may hold the same Class_names as node's labels, and the same attributes as relationships. Therefore, to efficiently model such type of data, the property (Model: Model_name) will be added to each node in the graph database, as depicted in Listing 4.5 below. Having the model name as one of the node's properties, will allow the user to use this property to define and retrieve all data that belong to the specific model, easily.

```
LOAD CSV WITH HEADERS FROM "file:///Muster003/IfcMaterial.csv" AS line
FIELDTERMINATOR' '
CREATE (material: IfcMaterial{IFCID:line.IFCID, Model:line.Model,
name:line.name})
```

Listing 4.5: Cypher command to automatically create nodes using CSV data

Having developed all nodes that represent objects belong to the IFC model under consideration. Relationships can be then created using these nodes. Here, it should be mentioned that two fundamental node's properties are used to specify which node should be linked with which node of the same model. These two node's properties are:

1. Model: Model_name
2. IFCID: Step-Id

Using one graph database for multiple IFC models offers different major advantages which can be summarized as following:

1. Co-exiting of several IOG models in one machine, allow the user to operate new connectivity between different models which can provide an efficient way to manage storage capacities.
2. Clash detection can be executed by loading IFC data of the same model from different disciplines.
3. Comparison study can be performed on IFC models of different versions to detect updates and indicate new additives, or even trace the history of the specific model by reporting changes.

The following Cypher query of Listing 4.6, will be used to detect any differences or variations between two different models, this query has been proposed earlier for illustration purposes only, and will be applied in the next chapter.

```
MATCH (a {Model: 'Muster003_V1'}), (b {Model: 'Muster003_V2'})
WITH [a] as New_Standard, [b] as Existing_ Standard
RETURN FILTER (n IN New_Standard WHERE NOT n IN Existing_ Standard) as
New_Items
```

Listing 4.6: Cypher command to detect variations between two IOG models

4.3.2 Multi-labeled entities

The second important enhancement is to extend the single classification of graph nodes to support a multi-classification., where each entity will have several classes (labels). Basically, the IOG models generated based on the idea that each entity within the graph database has a single label, which is the class name of that entity as introduced earlier. Here, each node is enriched by adding class names of all supertype classes based on the hierarchical structure. The key idea is that each

node label will hold the instance name followed by a series of class names of all supertype classes, this is achieved using the IFC Meta Graph model that introduced in Chapter 3 as a platform.

Addition of labels

A multi-statements Cypher script has been generated with the aim to add labels to the existing entities by making use of the SET clause of Cypher query language. The SET clause is normally used to update labels and properties on nodes and relationships. For the objective of the present section, the SET command will be used to add labels only as it is shown in the Listing 4.7 below, which an excerpt from the entire multi-statements Cypher script that been attached to Appendix A.5. Listing 4.7 illustrates the concept of multiple labels on the *IfcSlab* entity of the Muster003 model as an example.

```
MATCH (slab: IfcSlab {Model: 'Muster003.ifc'})
SET slab:IfcBuildingElement:IfcElement:IfcProduct:IfcObject:
IfcObjectDefinition:IfcRoot;
```

Listing 4.7: Cypher command to add labels to *IfcSlab* entities within IOG model

This set of multiple labels (*IfcBuildingElement*, *IfcElement*, *IfcProduct*, *IfcObject*, *IfcObjectDefinition* and *IfcRoot*) will be added to the entity *IfcSlab*. Therefore, each time one this entity is retrieved, this set of multiple labels will be shown in addition to the original data of the entity, as depicted in Figure 4.8 below.

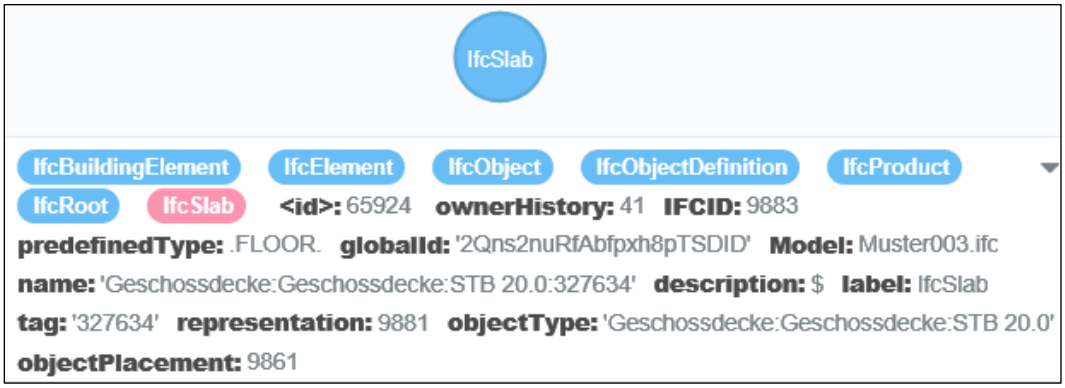

Figure 4.8: Graphical representation of the *IfcSlab* and its set of multiple labels

Filtering nodes based on multi-labels

Having multi-labeled entities will significantly facilitate filtering of information, where any group of IFC entities can easily be filtered by defining the superior class that combines those entities under consideration since that several entities will meet somewhere along the hierarchical path. For the modified model of Muster003, all building elements can be retrieved using single Cypher graph query command as it is stated in Listing 4.8 below.

```
MATCH (element: IfcBuildingElement {Model: 'Muster003.ifc'})
RETURN element
```

Listing 4.8: Cypher command to detect variations between two IOG models

Having applied this simple query in Neo4j graph database will ensure retrieving all entities which have the label IfcBuildingElement, as part of their multi-labels. The outcome of the building elements query is graphically represented by the graph database as it is shown in Figure 4.9 below.

The result demonstrates that there are 23 building elements belonged to the IFC model under consideration, which is indeed the number of building elements within the Muster003 model

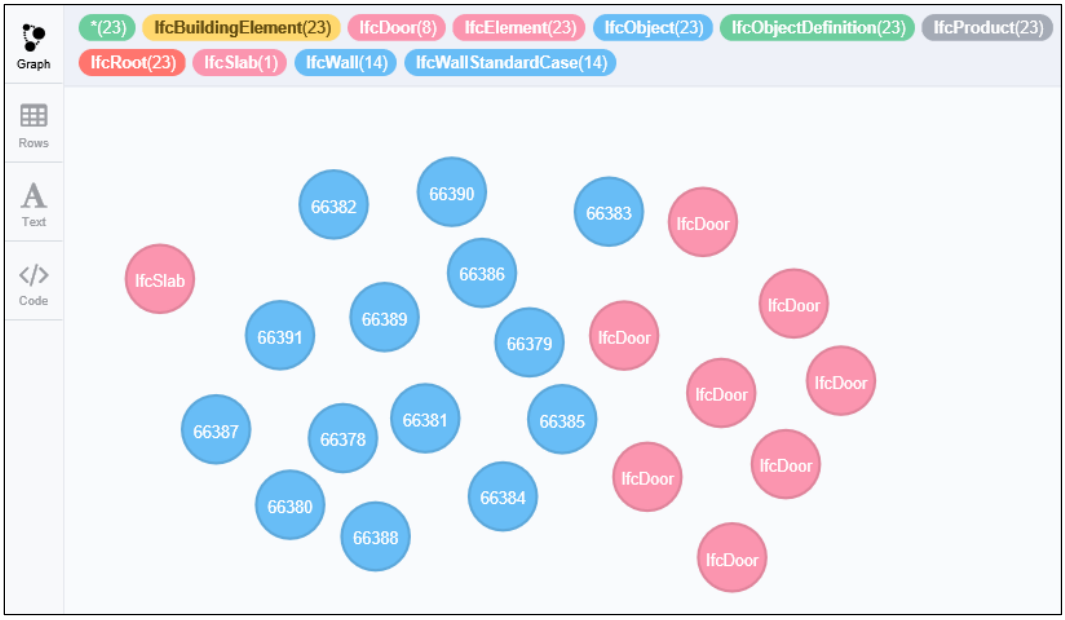

Figure 4.9: Filtering building elements using multi-labels technique

4.4 Experimental case studies

The work presented in this study is based on two BIM models as experimental cases. The first IFC data represent a single-storey building model known as ‘Muster003’ model. The model is used as a primary case to study the effectiveness of graph generation methodology that been introduced earlier at the beginning of the present chapter. This experimental model contains typical building elements such as spaces, walls, doors and slab. Where 5,938 entities are used to create the model as illustrated in the 3D Visualization of Figure 4.10 below.

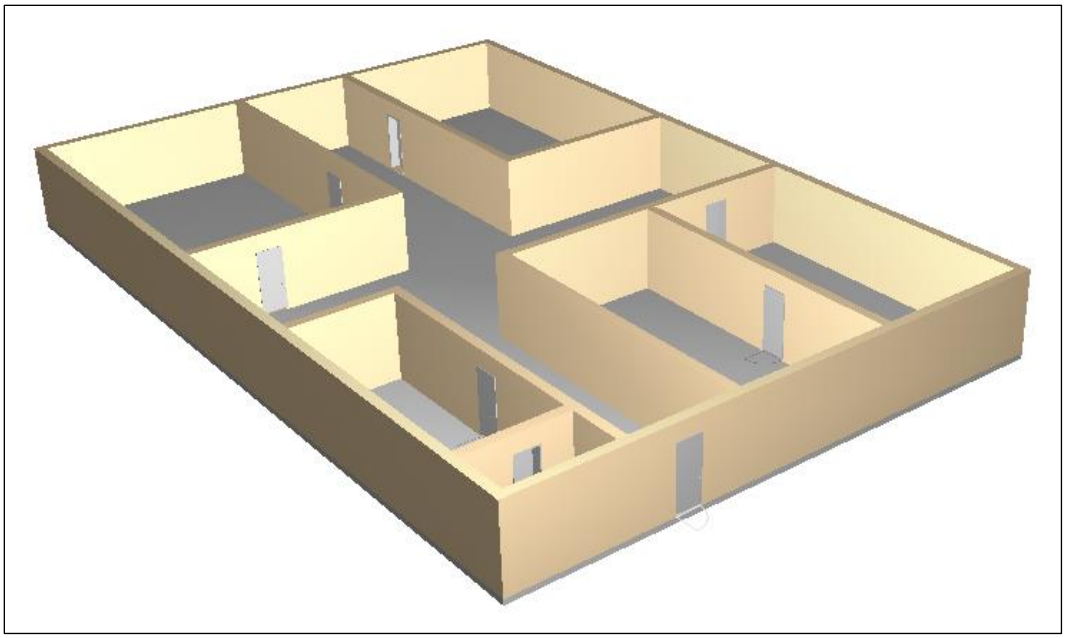

Figure 4.10: 3D Visualization of the 'Muster003' model

While the second IFC model is one of the buildingSMART Common BIM models (http://www.nibs.org/?page=bsa). It presents a three storey office building with a complex structure, which contains 62,930 entities to express the architectural building information model as shown in Figure 4.11. Thus, the names 'Muster003' and 'Office_A' will be used to refer to the first and second models through the whole study.

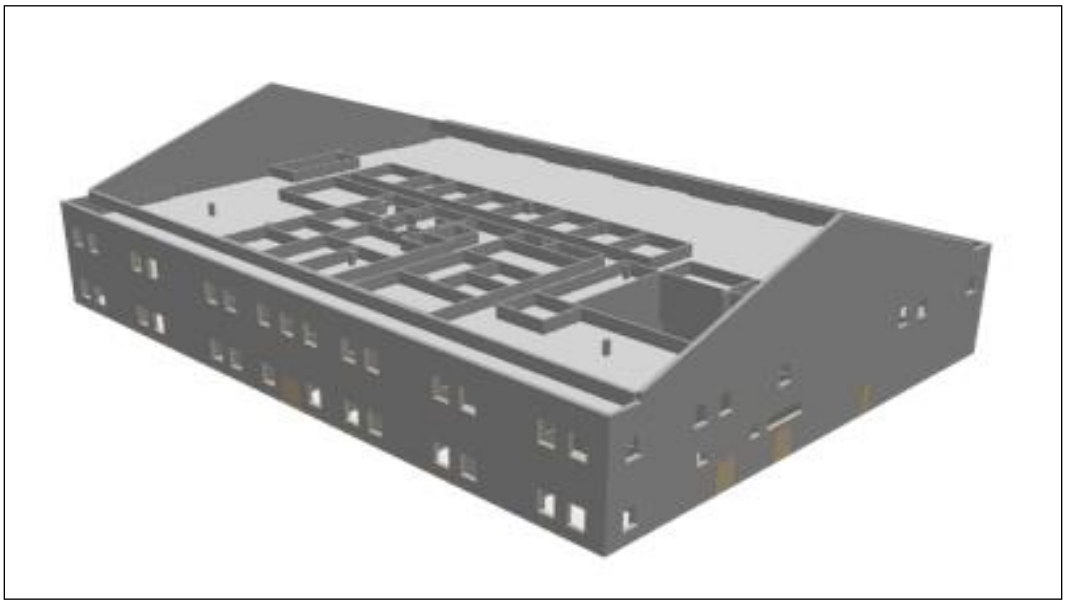

Figure 4.11: 3D Visualization of the 'Office_A' model

Chapter 5: Filters and Queries

5.1 Introduction

In this chapter, the potential of using graph database to explore, visualize and analyze BIM models is demonstrated by numerous examples. The examples address the application of filters and queries at three different levels in terms of complexity, to provide detailed information and graphical insights. The three different levels of filters implementation expressed in this chapter are not clear-cut categories. Therefore, each section contains several examples based on Cypher query language to merely serve the case under consideration. Figure 5.1 illustrates the levels of filters implementation as expressed earlier.

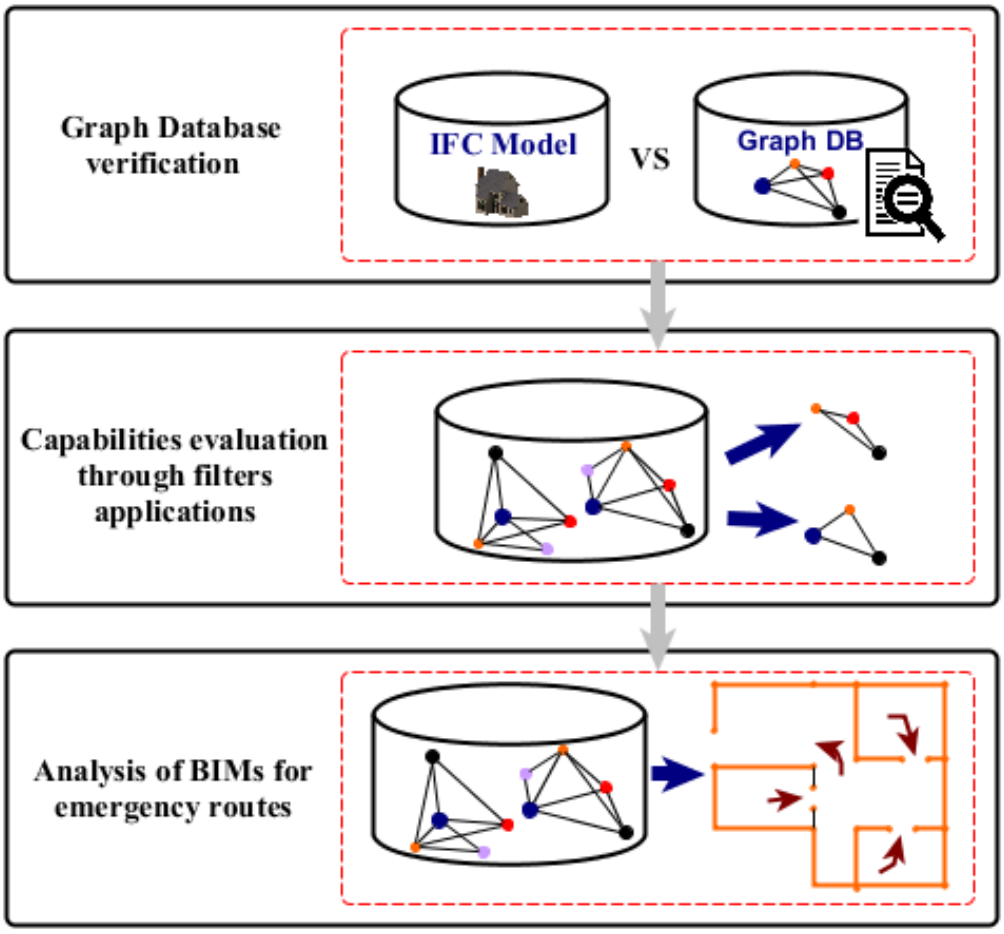

Figure 5.1: Levels of filters implementation

The first section introduces (Section 5.2) a graph database verification queries. Where simple queries are implemented to check the correctness of the generated IOG models against the original IFC model. The verification is performed to indicate the unintended changes that may occur during the data transmission due to data loss, data duplication or data corruption. Then, this step is

followed by execution of data modification of incorrect relationships and objects and the addition of missing data. While section 5.3 investigates the capabilities of graph database in answering realistic questions and retrieving building information that cannot be retrieved using normal BIM queries based on the IFC object model. Finally, advanced analysis of Building Information Models is performed in the specific area of building topology analysis which is indoor navigation route analysis. Where complex queries are applied as an example to find the possible emergency route by making use of space nodes and the connected accesses.

5.2 Graph database verification

The verification process is executed in this section to explore the variations and differences between the original IFC models, and the datasets that developed in the graphs database, as depicted in Figure 5.2 below. The graph database verification process has been divided into two main stages, it is initiated by comparison process for objects verification and then followed by relationships verification.

For the graph database verification, there is a need for a third-party system to be used to display the IFC data within the original STEP file, such as classes names, the number of entities, relationships, and associated attributes. Therefore, the online IFC data model server IFCWebServer that introduced earlier, will be used here again as a tool to explore IFC data, then these data are compared with the data extracted from graph models.

Furthermore, the necessary actions required to correct or modify the graph database in case of detecting any variations will be shown in this section, since that data transferring process could differ from a model to model.

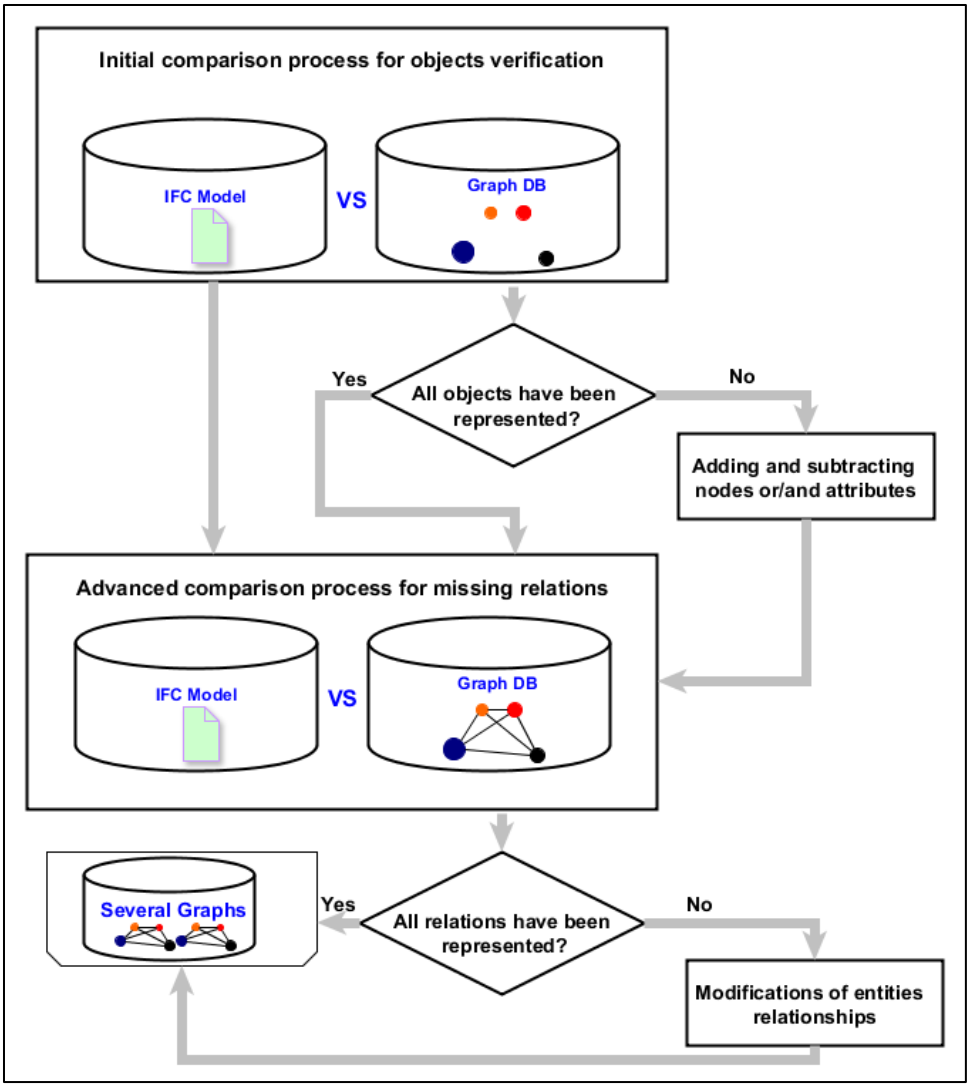

Figure 5.2: Graph database verification process

5.2.1 Initial comparison

Firstly, the IFC data inside the IFC file are expressed using the IFCWebServer. It provides interfaces and tools to simply load and analyze BIMs based on IFC standards. Thus, users can apply filters and queries on the loaded models, and report the search outcomes. Numerous function icons are provided as predefined queries to facilitate frequently search tasks, such as the structure of BIMs through IFC objects list. Then, original data from the IFCWebServer are compared with data retrieved from the graph database to validate the completeness of IFC to Graph DB mapping workflow.

The data profiling processes in this section start out by reviewing the total number of IFC entities within the IOG models. This task is accomplished by applying Cypher command in Listing 5.1 below, as an example to count the total number of entities within the ‘Muster003’ model.

```
MATCH (n {Model: 'Muster003.ifc'})
RETURN COUNT (n) AS total
```

Listing 5.1: Cypher command to count total number of IFC entities

The graph database has shown that there are 5,938 entities belong to ‘Muster003’ model as shown in the snapshot of Figure 5.3, which is an exact number of entities within the IFC file as illustrated by the IFCWebServer and shown here in Figure 5.5.

Figure 5.3: Total number of ‘Muster003’ model’s IFC entities by Neo4j

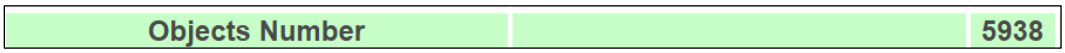

Figure 5.4: Total number of ‘Muster003’ model’s IFC entities by IFCWebServer

In the next data verification example, the class names list and their corresponding IFC entities stored within the graph database are reviewed. In fact, Neo4j graph database enables users to display node labels by using LABELS statement and count entities by simply applying COUNT statement as depicted in previous Cypher query example. As mentioned earlier, the IFC entities have been represented as nodes, labeled based on the class name they represent. For illustration purposes, an excerpt from the IFC class types and entities list, for ‘Muster003’ model is presented below. The list is acquired by applying Cypher command in Listing 5.2.

```
MATCH (n {Model: 'Muster003.ifc'})
WITH DISTINCT LABELS (n) AS LABELS, COUNT (n) AS temp
UNWIND LABELS AS Class_name
RETURN DISTINCT Class_name, SUM (temp) AS Number_of_entities
ORDER BY Class_name
```

Listing 5.2: Cypher command to retrieve object node list and count number of entities

Running Cypher query command in Listing 5.2 succeeds to return a list of classes names similar to classes names list in the original IFC ‘Muster003’ model, displayed using IFCWebServer. Table 5.1 illustrates an excerpt from IFC objects list that retrieved from the graph database using the previous Cypher command.

Table 5.1: Excerpt from Muster003's IFC objects list as retrieved by Neo4j

Class_name	Number_of_entities
IfcApplication	1
IfcArbitraryOpenProfileDef	57
IfcAxis2Placement2D	33
IfcAxis2Placement3D	153
IfcBuilding	1
IfcBuildingStorey	1
IfcCartesianPoint	2534
IfcCartesianTransformationOperator3D	1
IfcCircle	1
IfcClosedShell	244
IfcColourRgb	10
IfcConnectionSurfaceGeometry	66

The same list can be retrieved for 'Office_A' model by changing the property value 'Muster003.ifc' of the MATCH command into 'Office_A.ifc'. Having applied Cypher command of Listing 5.2 for 'Office_A' model, the graph database displayed the entire classes names list. However, Table 5.2 shows an excerpt from that list for illustration purposes.

Table 5.2: Excerpt from Office_A's IFC objects list as retrieved by Neo4j

Class_name	Number_of_entities
IfcApplication	1
IfcArbitraryClosedProfileDef	156
IfcArbitraryOpenProfileDef	1243
IfcArbitraryProfileDefWithVoids	22
IfcAxis2Placement2D	506
IfcAxis2Placement3D	1551
IfcBooleanClippingResult	4
IfcBuilding	1
IfcBuildingStorey	3

IfcCartesianPoint	12063
IfcCartesianTransformationOperator3D	1
IfcCircle	27

One of the data verification practices that can be executed here to ensure the quality of the produced IOG models, is to check absent of specific information such as unlabeled nodes, missing properties or even missing property values. The following Cypher query, which is shown in Listing 5.3, is used to return unlabeled nodes, while Cypher query of Listing 5.3 returns node only if they do not have any property value attached to them:

```
MATCH (n)
WHERE size(labels(n)) = 0
RETURN n
```

Listing 5.3: Cypher command to return unlabeled objects

```
MATCH (n)
WHERE size(keys(n)) = 0
RETURN n
```

Listing 5.4: Cypher command to return entities without properties

Cypher queries of Listing 5.3 and Listing 5.4 perform the traversal process to find entities that meet our information goals along the entire graph database. However, there is a possibility to anchor the search to a specific IFC model by adding the property key ‘Model’ and its value, this can improve the performance of the graph database.

5.2.2 Advanced comparison of IFC relationships

Although graphs are adaptable for modification and continuously evolving of datasets, it is still useful to review the models during the evolving process. Reviewing the relationships within the IOG models at an early stage could significantly avoid mapping errors or data loss during the import process, in addition, it supports retrieving correct information in advance stages. For this reason, queries are applied in this section to review relationships among entities, and to search whether there is any relationship has been missed or failed to be created, in compare to the original relationships in the IFC models. One of the used approaches is by reviewing IFC entities that have not been involved in any relationship, this can be achieved using the following Cypher query command in Listing 5.5:

```
MATCH (n {Model: 'Muster003.ifc'})
WHERE NOT ((n)--())
WITH DISTINCT LABELS (n) AS LABELS, COUNT (n) AS temp
UNWIND LABELS AS Class_name
RETURN DISTINCT Class_name, SUM (temp) AS Number_of_unconnected_entities
ORDER BY Class_name
```

Listing 5.5: Cypher command to return list of unconnected nodes (entities) and their number

After having applied Cypher query of Listing 5.5, the following list of unconnected entities in Table 5.3, has been retrieved. However, these relationships can be restored after diagnosing the original source behind the absent of these relationships.

Table 5.3: List of unconnected nodes (entities) and their corresponding number

Class_name	Number_of_unconnected_entities
IfcCurveStyleFont	1
IfcCurveStyleFontPattern	1
IfcDerivedUnit	2
IfcMaterial	3
IfcPresentationLayerAssignment	4
IfcPresentationStyleAssignment	16
IfcProductRepresentation	9
IfcPropertySingleValue	240
IfcSurfaceStyle	7
IfcCurveStyleFont	1

Moreover, entities-relationships within an IOG model can be investigated though the number of relationships themselves. The total number of relationships within an IOG model can be considered as a useful tool during the data correctness process, to review whether the missing or duplicated relationships have been successfully combined, removed or not, just by showing the total number of relationships after each modification step, instead of retrieving the entire list of relationships. The total number of relationships is retrieved by running Cypher query command in Listing 5.6 below.

```
MATCH (n {Model: 'Muster003.ifc'})-->()
RETURN COUNT(*) AS Total_number_of_relationships
```

Listing 5.6: Cypher command to count total number of relationships

Finally, it should be noted that Neo4j interface provides numerous function icons for users to execute data profiling tasks, such as a total number of nodes, relationships, a list of node labels. However, such kind of icons could not be useful in case of multiple graph models within the same database, since that the search cannot be limited to a specific model.

5.3 Graph capabilities evaluation

The purpose of the present section is to investigate the capability of IOM models in answering queries. However, the IOG models in this study have been designed to answer questions in form of Cypher queries, therefore, the following query and filter examples are applied, firstly, to demonstrate the flexible relationships that offered by graph networks in comparison with complex EXPRESS data model schema of IFC. And secondly, to evaluate the capability of the IOG models in exploring the relationships inside BIMs. Therefore, several graphs query examples will be applied on the two building models (Muster003 and Office_A) that presented earlier as study cases.

For each filter case within this section, the graph capability evaluation process initiates by expressing IFC class sub-diagram that could support understanding of the query case under consideration and then followed by a comparison of query's outcomes with query's results obtained using the online IFC data model server IFCWebServer.org.

5.3.1 Example (1): Retrieve the assigned properties of a certain object

In the following query example, the graph database will be queried with the objective to retrieve objects based on their property or vice versa. In fact, the IFC specification assigns each property set *IfcPropertySet* to its target objects *IfcObject* using the objectified relationship *IfcRelDefinesByProperties*. While, individual properties of each property sets can be described separately using a single value property *IfcPropertySingleValue* object, which has single numeric or descriptive value. Figure 5.5 illustrates the relationship between different elements *IfcElement* such as the *IfcDoor, IfcSlab, IfcStair* and *IfcWindow*, and the property set.

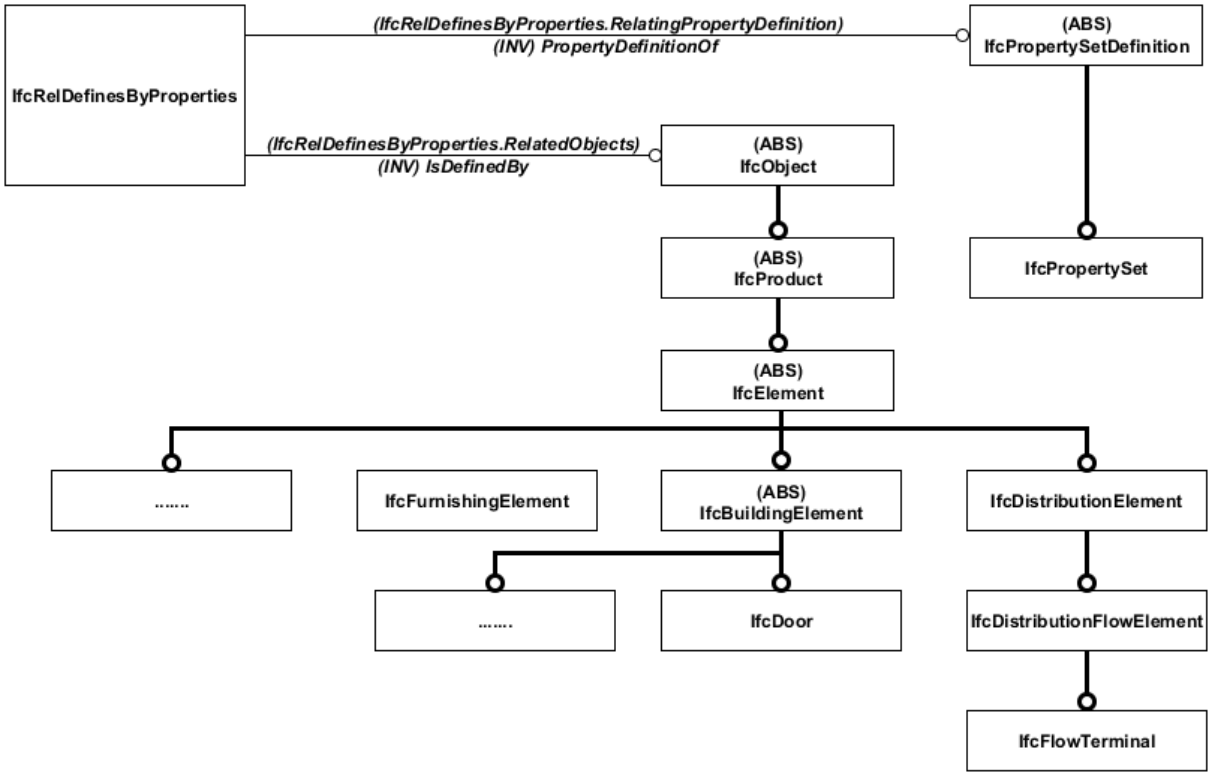

Figure 5.5: IFC class diagram describing the element-property set relationship

The knowledge that gained from the previous IFC class sub-diagram regarding the Object-Property relationship, this knowledge can be used to filter specific element or group of elements from the graph database based on their property, or vice versa, basic information such as property names and values with measure type can be retrieved based on the element or object under study. However, the above IFC classes relationships are represented in the graph database with some differences, which can be summarized as following:

1. All the abstract classes are not considered within the IOG models, while, the non-abstract classes are represented as nodes.
2. A unique relationship *IsDefinedByProperties* has been created to link property sets directly with objects during the graphs development process. This relationship enables the user to pick out entities directly, rather than discover them over other complex relationships.

The following query example searches out for property sets which have been used to define a specific object by making use of the *IsDefinedByProperties* relationship above. As it is shown in Listing 5.7, which states a Cypher query to retrieve the property sets that define the walls with 'IFCID=2091' within 'Muster003' model.

```
MATCH (wall: IfcWallStandardCase{Model:'Muster003.ifc',IFCID:'2091'})-
[rel]-(property: IfcPropertySet)
RETURN DISTINCT wall,rel,property
```

Listing 5.7: Cypher command to retrieve elements based on property set's name 'Tragwerk'

However, the property sets define the properties of the *IfcWallStandardCase* in Listing 5.7 are graphically represented by Neo4j as shown in Figure 5.6, while their IFCIDs are listed in Table 5.4.

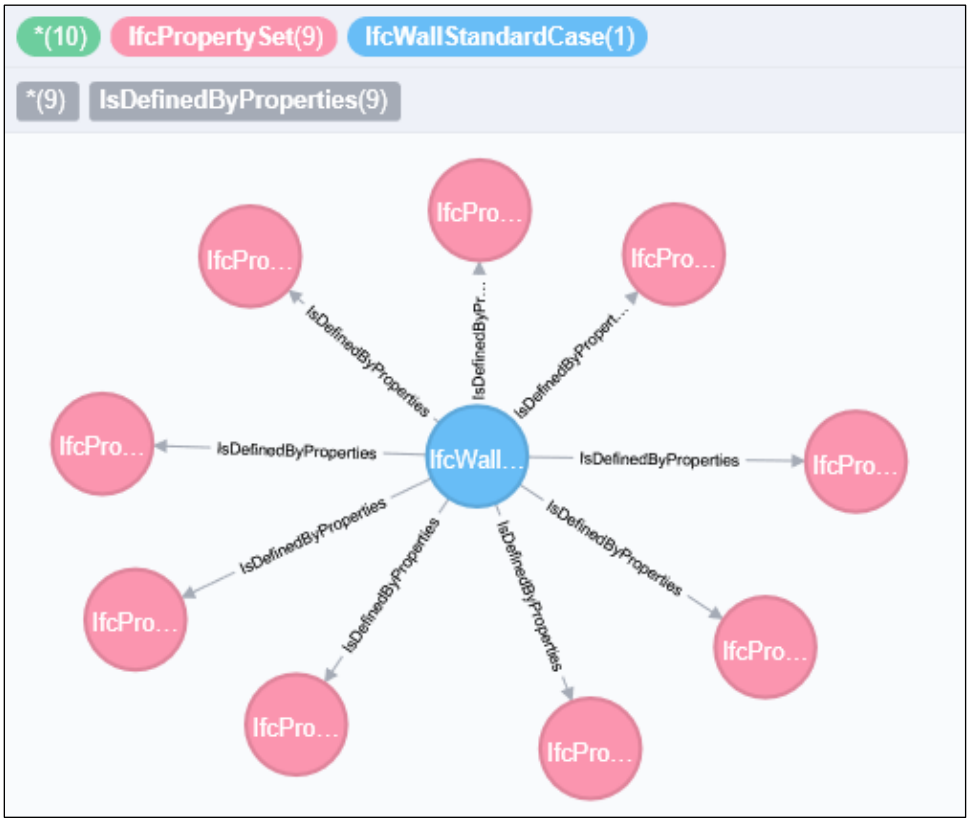

Figure 5.6: Graphical representation of property sets define specific wall

Table 5.4: Property sets define specific wall within Muster003 IOG model

No.	1	2	3	4	5	6	7	8	9
IFCID	2139	2134	2146	1427	2122	2129	1425	1431	1429

Vice versa, graph users can easily retrieve objects by defining property value and then return the element which is defined by this property value. In the following example, elements with a property set's name 'Tragwerk' are retrieved. To perform such query, the Cypher command of Listing 5.8 is executed.

```
MATCH (element {Model: 'Muster003.ifc'})-[rel: IsDefinedByProperties]-
>(property: IfcPropertySet {IFCID:'1565',name:'Tragwerk'})
RETURN DISTINCT element,rel,property
```

Listing 5.8: Cypher command to retrieve elements with property set "Tragwerk"

Having the query command of Listing 5.9 executed, the database returns the following elements which are defined as 'Tragwerk'. The query outcome is depicted in Figure 5.7 below.

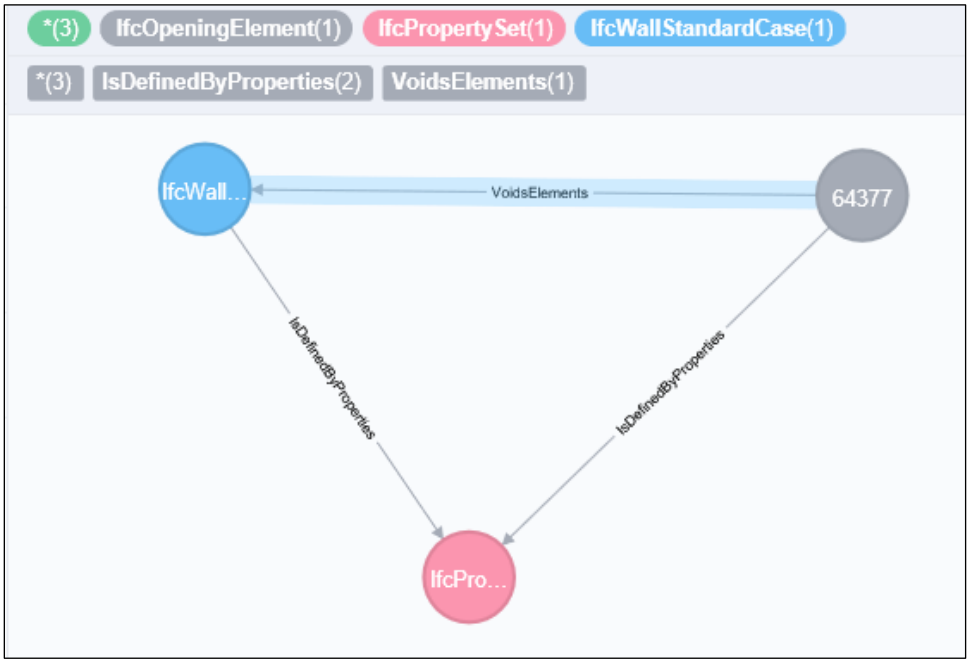

Figure 5.7: Elements defined by the property set's name 'Tragwerk'

5.3.2 Example (2): Filter objects based on material

In the IFC database, the non-virtual elements are associated with their material *IfcMaterial* instances through *IfcRelAssociatesMaterial* relationships. Where the *IfcMaterial* instances are utilized to describe homogeneous or inhomogeneous substances that are used to make or manufacture these elements or their components. From the IFC class diagram of Figure 5.8 below, it is easy to deduce that the material lists *IfcMaertialList* for each group of instances made from the same material, are placed couples of relationships away from the objects. Again, the graph database allows one to apply queries and retrieve information based on the relationship path without defining most of the intermediate nodes, relationship types or relationship directions, along with the path.

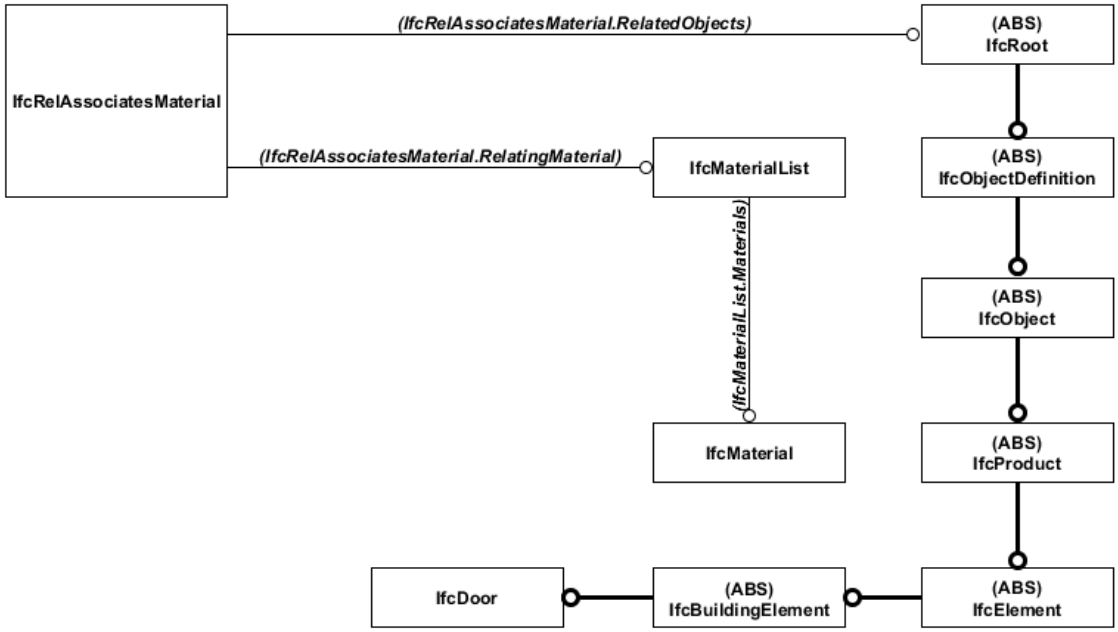

Figure 5.8: IFC class diagram describing door-associated material relationship

For example, to return a list of materials *IfcMaterial* that associated with door elements *IfcDoor* in the 'Muster003' model, the Cypher query of Listing 5.9 below, can be applied in Neo4j graph database to achieve this goal:

```
MATCH (door: IfcDoor {Model: 'Muster003.ifc'})-[*1..5]-(material:
IfcMaterial)
RETURN DISTINCT (material.IFCID) AS IFCID, (material.name) AS Material_name
```

Listing 5.9: Cypher command to retrieve list of materials associated with specific element

The above query in Listing 5.9, has been used to return the list of the materials that associated to door elements in the Muster003 model, with their IFCD, as shown in Table 5.5

Table 5.5: The Returned list of materials associated with doors

IFCID	Material_name
9472	'Metall - Lackiert - Grau'
9471	'Holz - Birke'
9470	'Metall - Chrom'

However, to specify which material has been used for which door within the model, then one of the unique properties which support to distinguish the targeted door from other accesses in the model, this property should be added to MATCH command of Cypher query in Listing 5.9. Thus,

Listing 5.10 expresses the modified graph query script to retrieve the materials that associated with the specific door, in this case, the door with property key 'IFCID' which it is value '9495', will be shown for illustration purpose.

```
MATCH (door: IfcDoor {Model: 'Muster003.ifc', IFCID: '9495'})-[*1..5]-
(material: IfcMaterial)
RETURN DISTINCT (material.IFCID) AS IFCID, (material.name) AS Material_name
```

Listing 5.10: Cypher command to return list of materials associated with specific space

The graph query had returned the same list of material as in Table 5.5, which means that all the doors within the IFC model under investigation, are having the same material list *IfcMaertialList*. However, by displaying the entire relationship between the elements and their associated material this can easily be checked. By applying Cypher command in Listing 5.11 below, the entire relationship can be exhibited as a graph, as depicted in Figure 5.9.

```
MATCH p=(door: IfcDoor {Model: 'Muster003.ifc', IFCID: '9495'})-[]-(relDef:
IfcRelDefinesByType)-[]-(doorSt: IfcDoorStyle)-[]-(relAss:
IfcRelAssociatesMaterial)-[]-(matList: IfcMaterialList)-[]-(material:
IfcMaterial)
RETURN p
```

Listing 5.11: Cypher command to display materials associated to specific space

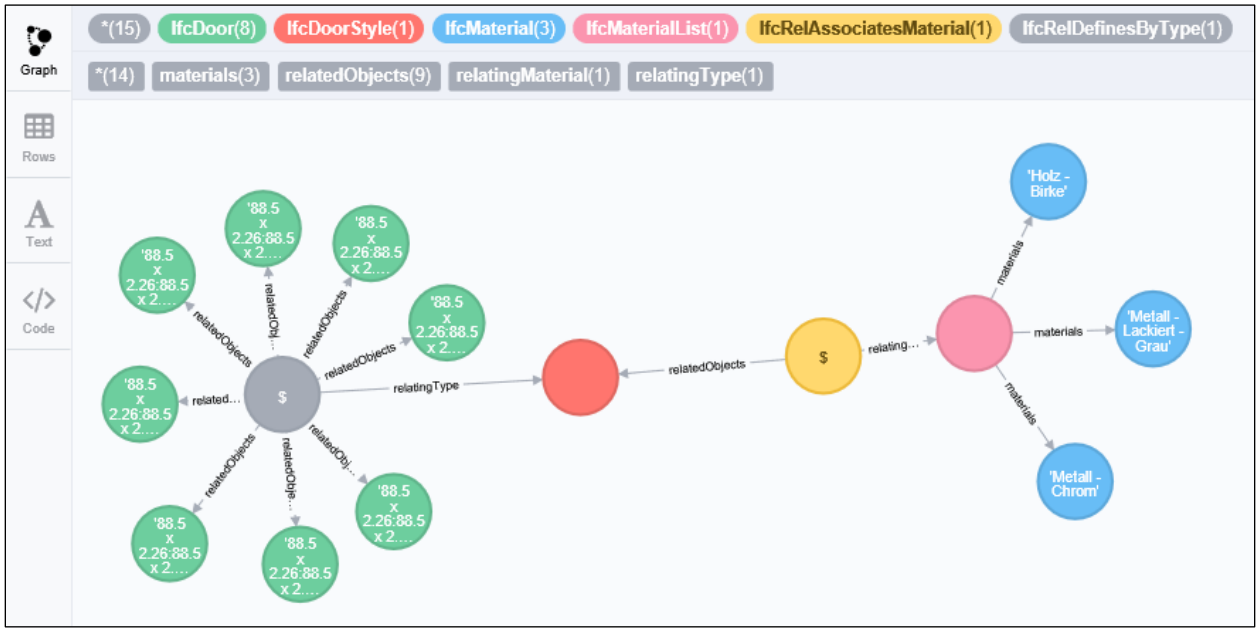

Figure 5.9: Displaying doors retrieved based on material type using Neo4j

Now, it is clear that all the eight doors within the 'Muster003' model, belong to the same door style *IfcDoorStyle* and made out of the same materials (the three materials that listed in Table 5.5)

5.3.3 Example (3): Filter objects based on aggregation relationship

The IFC specification defines the relationship between the objects and their subtype objects using the aggregation relationship *IfcRelAggregates*, which is considered as a special branch of the general decomposition/composition relationship *IfcRelDecomposes*. For example, the stair is an aggregation of several elements such as stair flights, stair carriages, railings, etc. The IFC class diagram in Figure 5.10, describes the aggregation relationship between the building stories and their related spaces.

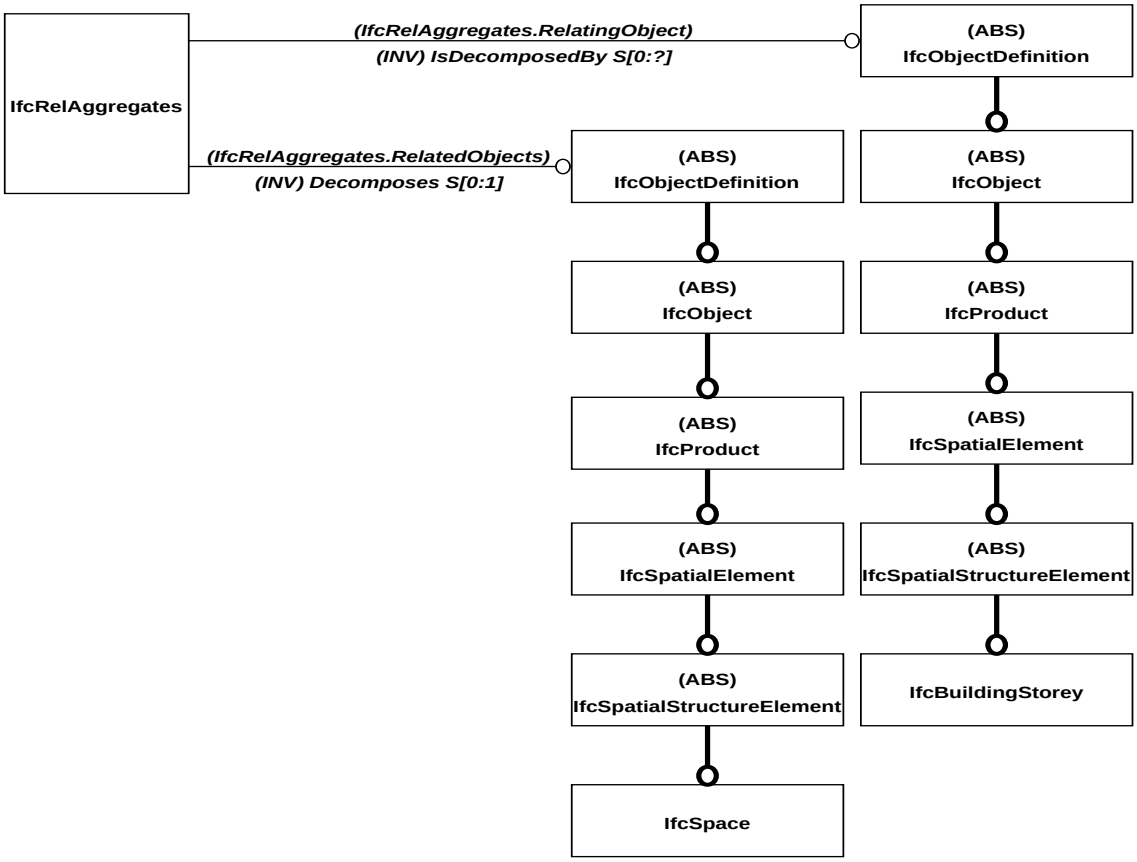

Figure 5.10: IFC class diagram describing space-building storey relationship

In our graph database, if we are looking for spaces that located within a certain building storey. The intended spaces and the building storey are represented within the graph database as nodes labeled *IfcSpace* and *IfcBuildingStorey* with 'IFCID' as properties for each node. Therefore, to return all the spaces *IfcSpace* within a certain building storey *IfcBuildingStorey* the below Cypher command of Listing 5.12 could be used to achieve this aim.

```
MATCH (space: IfcSpace{Model: 'Muster003.ifc'})-[]-(storey:
IfcBuildingStorey)
RETURN DISTINCT space.IFCID AS IfcSpace
```

Listing 5.12: Cypher command to return spaces located within a certain building storey

Adding this query snippet into Neo4j editor means that the database checks all the relationships between the *IfcSpace* nodes and the matched *IfcBuildingStorey*. Matches with *IfcSpace* nodes which place two relationships - one outgoing and anther incoming - away from the node that representing the building storey *IfcBuildingStorey* will pass the test; these spaces will be shown as results. However, matches that miss the test will not be part of the outcomes. Running Cypher command in Listing 5.12 will return all spaces that have been located within the building storey that specified earlier, the intended spaces are shown in Table 5.6.

Table 5.6: Spaces located within a certain floor

No.	1	2	3	4	5	6	7	8	9
IfcSpace (IFCID)	755	874	626	507	378	249	117	1201	1007

Here, it is important to mention that the study case in this study contains only one building storey. But in the case where a multi-storey building is investigated, then there is need to specify the named storey by adding a property to *IfcBuildingStorey* node in Listing 5.13. This listing illustrates an example of Cypher query script to return spaces belong to the specific storey.

```
MATCH (space: IfcSpace {Model: 'Muster003.ifc'})<-[rel1]-(n)-[rel2]-
>(storey: IfcBuildingStorey)
RETURN DISTINCT space.IFCID AS IfcSpace
```

Listing 5.13: Cypher command to return spaces located within a certain building storey

To get all spaces on each floor for the ‘Office_A’ model, the Cypher command of Listing 5.15 can be executed. The outcomes are graphically represented in Figure 5.11 below.

```
MATCH p=(storey: IfcBuildingStorey{Model:'Office_A.ifc'})-[rel:Decomposes]-
(space: IfcSpace)
RETURN p
```

Listing 5.14: Cypher command to return spaces located within a certain building storey

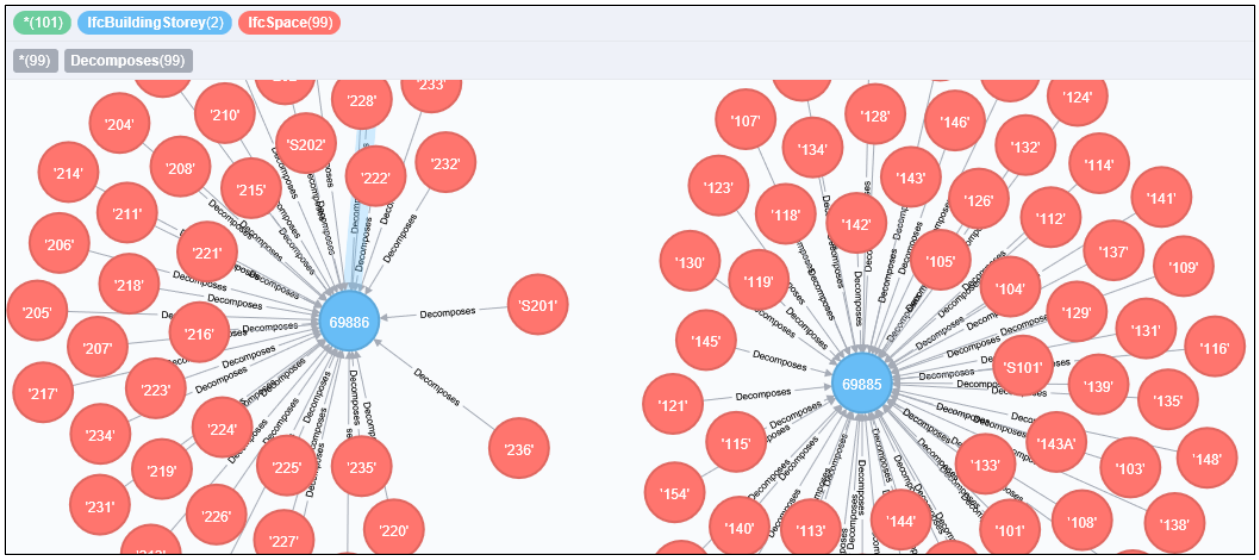

Figure 5.11: Graphical representation of building stories with spaces

5.3.4 Example (4): Filter surrounding elements

In this section, we are going to study the relationship between the physical space boundaries and the space that they define. For instance, building elements such as walls, doors, and slabs that surround specific space are introduced here as an example. The intended space is represented within the graph database as a node labeled *IfcSpace* with 'IFCID' as property key whose value is '874'. The property value could be bound to the identifier 'a', this identifier allows us to refer to the node that represents this particular space throughout the rest of the queries.

First, to understand how spaces are connected to their surrounding building elements within the IFC database, it is important to explore the IFC diagram that defines such relationships. The IFC specification links the *IfcSpace* objects with their related building elements that surrounded this space, through the *IfcRelSpaceBoundary* objects. In this case, the physical space boundary may be given by building elements as it is depicted in Figure 5.12.

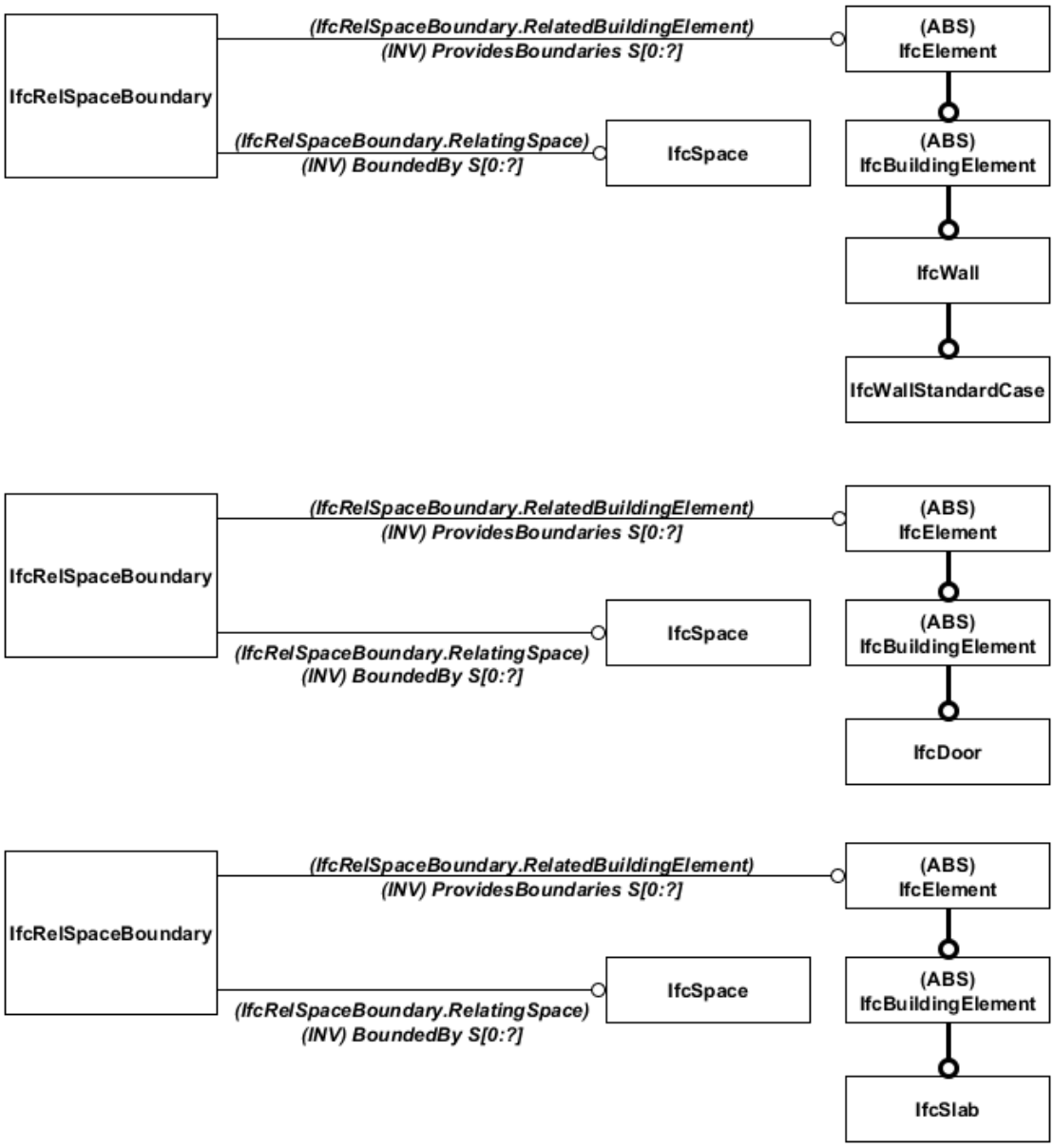

Figure 5.12: IFC diagrams for space-surrounding building elements relationships

The following Cypher command in Listing 5.15, Listing 5.16 and Listing 5.17 can be executed respectively to return all type of walls, doors, and slabs that surround the space with (IFCID:874):

```
MATCH (space: IfcSpace {Model: 'Muster003.ifc', IFCID:'874'})
MATCH (space)<-[*1..2]->(wall: IfcWallStandardCase)
RETURN COLLECT(wall.IFCID) AS IfcWallStandardCase, COUNT(wall) AS
Number_of_walls
```

Listing 5.15: Cypher command to return walls that surround space with (IFCID: 874)

```
MATCH (space: IfcSpace {Model: 'Muster003.ifc', IFCID:'874'})
MATCH (space)-[*1..2]-(door: IfcDoor)
RETURN COLLECT(door.IFCID) AS IfcDoor, COUNT(door) AS Number_of_doors
```

Listing 5.16: Cypher command to return doors to access the space with (IFCID: 874)

```
MATCH (space: IfcSpace {Model: 'Muster003.ifc', IFCID:'874'})
MATCH (space)-[*1..2]-(slab: IfcSlab)
RETURN COLLECT(slab.IFCID) AS IfcSlab, COUNT(slab) AS Number_of_slabs
```

Listing 5.17: Cypher command to return slabs surround space with (IFCID: 874)

The following outcomes in Table 5.7, Table 5.8 and Table 5.9 are acquired as a result of executing Cypher queries that expressed earlier in Listing 5.15, Listing 5.16 and Listing 5.17 respectively:

Table 5.7: Walls that surround space with (IFCID: 874)

IfcWallStandardCase	Number_of_walls
[1726, 1668, 2091]	3

Table 5.8: Doors to access the space with (IFCID: 874)

IfcDoor	Number_of_doors
[9495, 9559]	2

Table 5.9: Slab connected to the space with (IFCID: 874)

IfcSlab	Number_of_slabs
[9883]	1

However, the search outcomes that demonstrated in previous tables, using command statements of Listing 5.15, Listing 5.16 and Listing 5.17 could not effective in the case of large disparate datasets, because several queries should be executed separately. Therefore, the following graph queries in Listing 5.18 has been proposed to overcome the issue of repetition, and enable graph queries user to return surrounding building elements by using single Cypher statement one at a time, as following:

```
MATCH (space: IfcSpace {Model: 'Muster003.ifc', IFCID:'874'})
MATCH (space)<-[*1..2]->(wall: IfcWallStandardCase), (space)-[*1..2]-(door:
IfcDoor),(space)-[*1..2]-(slab: IfcSlab)
RETURN (slab.IFCID) AS IfcSlab,(door.IFCID) AS IfcDoor, COLLECT(wall.IFCID)
AS IfcWallStandardCase
```

Listing 5.18: Cypher command to return boundary elements that surround specific space

The Cypher command investigate the physical space boundaries once and retrieve the following outcomes as shown in Table 5.10.

Table 5.10: Walls, doors, and slabs that surround space with (IFCID: 874)

IfcSlab	IfcDoor	Number_of_slabs
9883	9559	[1726, 1668, 2091]
9883	9495	[1726, 1668, 2091]

5.3.5 Example (5): Comparison IFC model versions

In this example, the differences between two versions of the same IOG model is demonstrated. This method can be used to trace the changes during building design. Whenever any change happened, one of the models will contain objects that the other one does not. Therefore, there is a need for comparison tools to be able to indicate the differences between any two different IFC models and show up the new or the modified information. Solibri Model Checker (SMC) [28] software provides a tool to compare two versions of the same model by relying on IFC building models as input. To trace and report details results of what have been added, removed or adjusted by means of their identification data, amounts, positions, and property. Another comparison system has been invented by Sohn [29], the system is created to manage IFC versions by comparing two or more IFC files generated at different time.

In the present work, two different versions of the "Muster003" model are used as case study, these two versions will be compared to show up the differences. The outcome is represented in Figure 5.13, which demonstrates that the name of the space no.4 has been changed in the new version from "4" to "Room 4".

s1.label	s1.globalId	old_name	new_name
IfcSpace	'07eMeFvu91xw9g0oWgvYEc'	'4'	Room 4

Figure 5.13: Comparison study of two versions of IOG models

The Neo4j graph database has the capability to match two disconnected graphs in the database, by matching entities of the two models under investigation and returning only entities which do not exist on the other model as depicted by WHERE clause of Cypher query in Listing 5.19 below.

```
MATCH (s1:IfcSpace{Model:"Muster003.ifc"}),(s2:IfcSpace
{Model:"Muster003_V2.ifc"})
WHERE s1.globalId = s2.globalId AND s1.name <> s2.name
RETURN s1.name AS old_name, s2.name AS new_name
```

Listing 5.19: Cypher command to trace new items in a modified model

5.4 Analysis of BIMs for emergency routes

In this section, the graph database system is utilized to develop emergency routes for the BIMs case studies in this study. Several studies have been accomplished in the last decade for analyzing building information, in general, based on data extracted from the IFC data model [30] [31]. However, using data extracted from IFC data models for a special area of research such as path planning or evacuation routes analysis is still rarely. Lin [32] developed an approach for IFC object path planning by obtaining the geometric and semantic information for 2D floors drawings or building layouts. While, other researchers proposed using graph representation of building information models based on IFC file as well, such as the study of Skandhakumar [33]. The study developed a graph model for indoor navigation applications in buildings to study the potential of using BIM graph in real-life applications, such as access control applications and facility management. In the present section, the graph theory is utilized to develop a graphical representation for the possible emergency routes as displayed in Figure 5.14 below. This graphical representation is gained as result of graph queries execution on the IOG models that generated in Chapter 4.

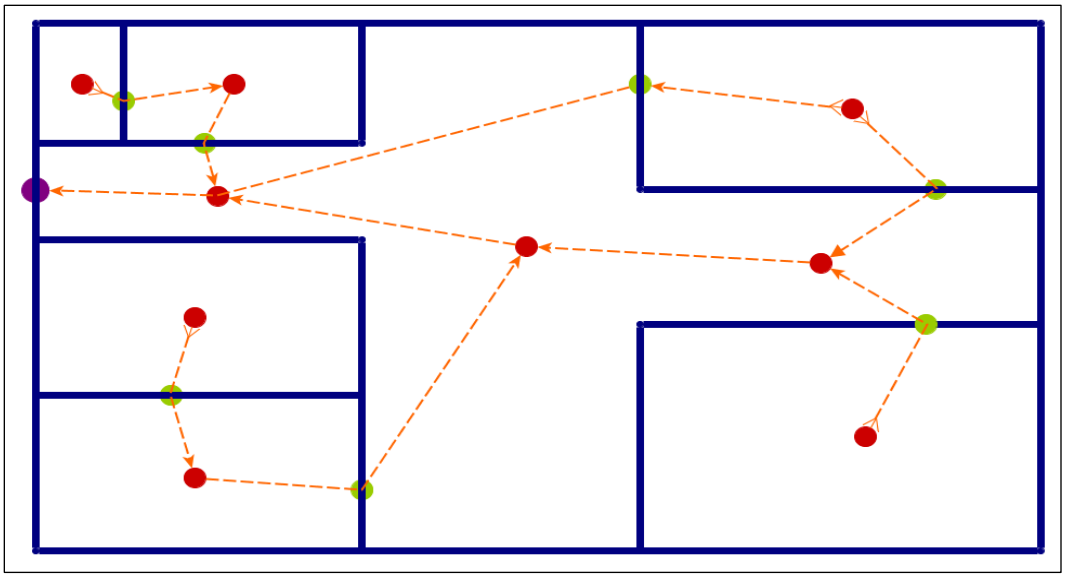

Figure 5.14: Graphical representation of emergency routes

5.4.1 Construction of possible paths

Extraction of information from a complex and highly connected dataset is quite challenging in general. However, in this section, the IOG models that generated earlier will be analyzed and queried to provide the possible emergency path networks for each floor, and then to link different building stories to create the vertical path using the geometric information that extracted from the IFC file.

1. Specify exit door as end nodes

As a starting point, we need first to specify the emergency exit door for zone or floor under study, the zone contains several spaces. Therefore, for each possible path, the space under consideration will be defined as the starting node, while the emergency exit door is defined as an end node. There will be several transmit nodes in between, depend on how far is the space from the emergency exit door. The transmit nodes are the other spaces and accesses along the passageway of the intended space. In fact, Cypher query allows the user to introduce starting point in case of having multiple relationship paths and anchor each path to start or end by that node as following Listing 5.20:

```
MATCH (node: IfcDoor {Model:'Muster003.ifc', name:'Exit door'})
RETURN node AS Exit_door
```

Listing 5.20: Cypher command to specify emergency exit door as end node for paths

2. Door-Space connectivity

To find the routes that connect each space with the exit door defined earlier, first, we are going retrieve spaces that sharing the same access. When two spaces are sharing the same door then it means, one of the spaces working as a path to another space. Then, we filter spaces

Access connecting spaces concept determines which spaces can be accessed through another space or spaces. This can be accomplished making use of the *IfcRelSpaceBoundary* relationship as shown in Listing 5.21 below:

```
MATCH (space1: IfcSpace{Model:'Muster003.ifc'})<-[rel1: BoundedBy]-
(door:IfcDoor)-[rel2: BoundedBy]->(space2: IfcSpace)
WHERE space1.IFCID > space2.IFCID
RETURN  space1.name AS FirstSpace, space2.name AS SecondSpace, door.IFCID
AS The_connecting_door
```

Listing 5.21: Cypher command to return connecting spaces and their shared access

Running Cypher query in Listing 5.21 against the 'Muster003' model, returns the connecting spaces and their shared access, as shown in Table 5.11.

Table 5.11: The retrieved connecting spaces and their shared access

FirstSpace	SecondSpace	The_connecting_door
'7'	'1'	9495
'7'	'2'	9559
'2'	'8'	9612
'4'	'3'	9665
'5'	'9'	9718
'3'	'8'	9771
'6'	'5'	10062

```
MATCH p=(space1: IfcSpace{Model:'Muster003.ifc'})-[rel1: BoundedBy]-(door:
IfcDoor)-[rel2: BoundedBy]-(space2: IfcSpace)
WHERE space1.IFCID > space2.IFCID, RETURN p
```

Listing 5.22: Cypher command to return navigation routes using connecting door

By graphically representing the outcomes of the above table using Cypher query of Listing 5.22 to return paths. it can easily be deduced that there is only two routes exist, due to fact that the traversal process moves from one space to another through the connecting doors only, but when there are no longer shared access the traversal process ceases. These two routes are graphically represented using Neo4j as in Figure 5.15 and sketched for illustration purposes as in Figure 5.16.

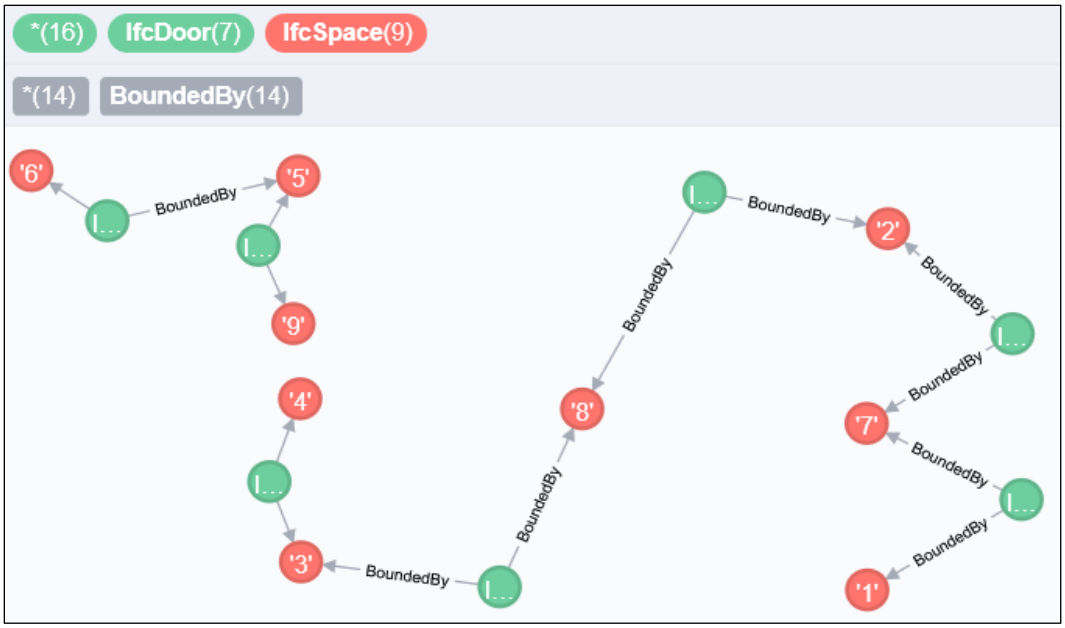

Figure 5.15: Graphical representation of door-space connectivity navigation routes

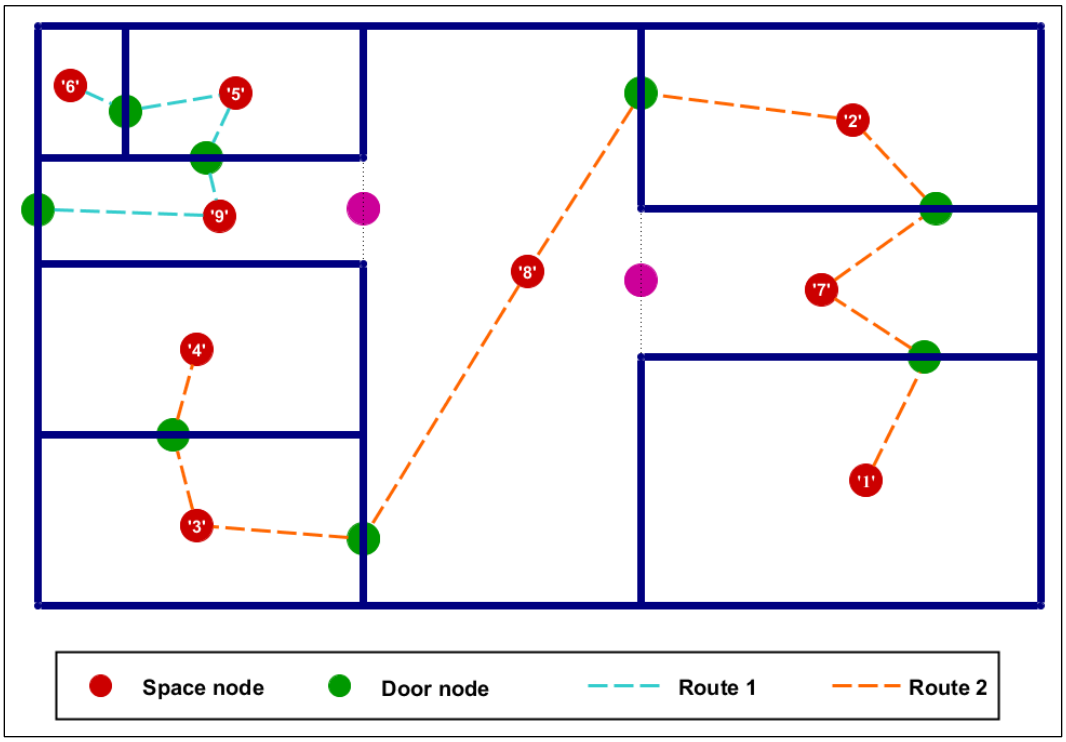

Figure 5.16: Graphical drawing of door-space connectivity navigation routes

3. Space-Space connectivity

It is very clear that, using of door-space connectivity cannot deduce logical paths, because navigation among adjacent spaces is possible only if there is a shared door, but navigation between

adjacent spaces is possible in many other cases, as in the cases where two spaces are open to each other, for instance, two adjacent corridors in a building. Here, one can make use of the *IfcRelSpaceBoundary* objects and its relationships, to created a new scenario for indoor navigation. The new scenario to enquire indoor routes has been applied in the graph database using the graph query in Listing 5.23 below.

```
MATCH p=(space1: IfcSpace {Model:'Muster003.ifc'})<-[rel1: RelatingSpace]-
(spaceBou: IfcRelSpaceBoundary)-[rel2: RelatingSpace]->(space2: IfcSpace)
WHERE space1.IFCID > space2.IFCID
RETURN p
```

Listing 5.23: Cypher command to return navigation routes using space boundary objects

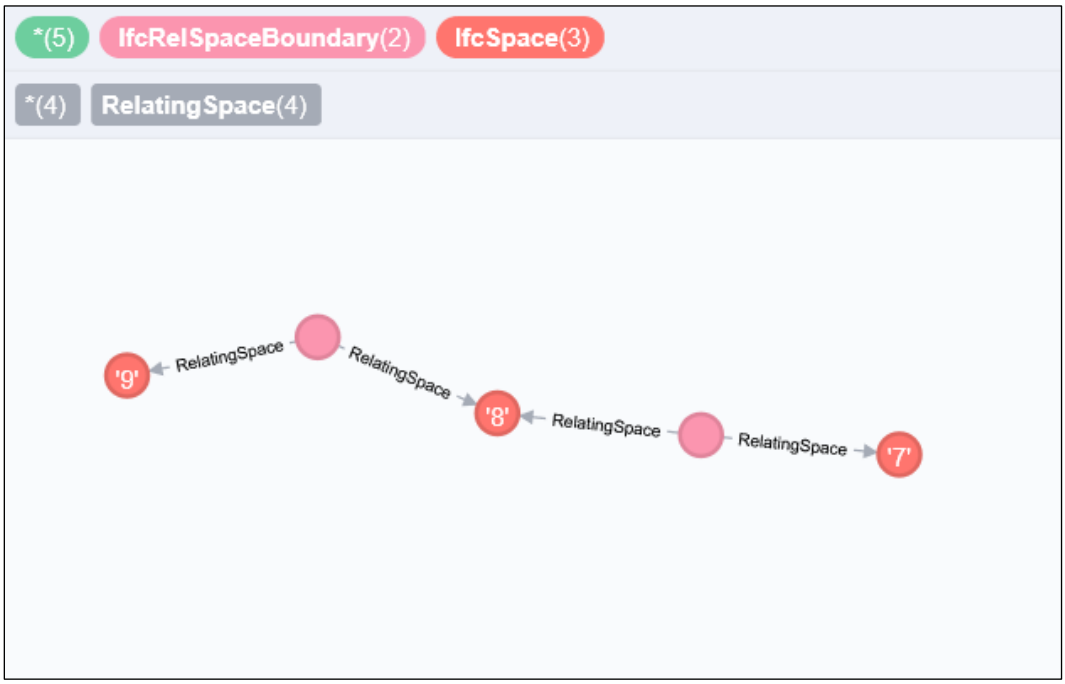

Figure 5.17: Graphical representation of space-space boundary connectivity navigation routes

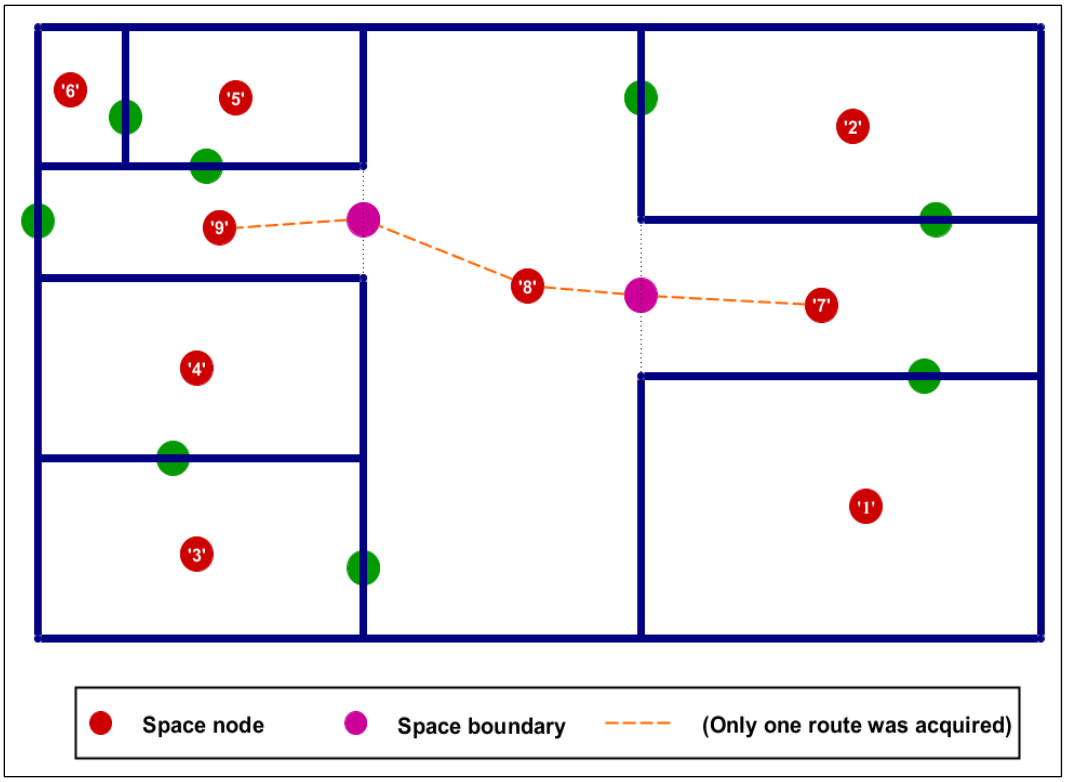

Figure 5.18: Graphical drawing of space-space boundary connectivity navigation routes

5.4.2 Retrieval query for emergency routes

Up to a point, it is very clear that utilizing one of the above query scenarios alone cannot achieve the objective since each scenario provides partial information regarding the complete emergency routes. While making use of graph power by combining several queries can serve the purpose. However, multiple queries can be combined using UNION clause of Cypher. This clause is normally used when there is need to merge several queries with similarly structured results. In our case, the three different queries that have been introduced earlier to achieve the following partial objectives:

1. Define the emergency exit door.
2. Specify indoor navigation routes through accesses.
3. Demonstrate navigation paths using space boundary.

All those queries will be combined in one query using the UNION clause as it is shown in Listing 5.24 below, to provide the entire emergency route which links each space within the storey under study with the emergency exit.

```
MATCH p=(door: IfcDoor {IFCID:'9824', Model:'Muster003.ifc'})-
[rel:BoundedBy]->(space: IfcSpace)
RETURN p
UNION
MATCH p=(space1: IfcSpace{Model:'Muster003.ifc'})<-[rel1: BoundedBy]-
(door:IfcDoor)-[rel2: BoundedBy]->(space2: IfcSpace)
WHERE space1.IFCID > space2.IFCID
RETURN p
UNION
MATCH p=(space1: IfcSpace {Model:'Muster003.ifc'})<-[rel1: RelatingSpace]-
(relSpace: IfcRelSpaceBoundary)-[rel2: RelatingSpace]->(space2: IfcSpace)
WHERE space1.IFCID > space2.IFCID
RETURN p
```

Listing 5.24: Cypher query to return emergency routes for entire floor

The above Cypher query in Listing 5.24 manages to connect each space located on the floor under consideration with the exit door, as it is depicted clearly in the graphical presentation of Figure 5.19 below. Moreover, if there is space with more than one exit such as in the case of space no. '7', then several routes will be demonstrated based on the number of the connected accesses. Here, it is important to specify which path is the shortest path to the exit door. Thus, Cypher provides a command to determine the shortest path between two points based on the number of relationships. However, the SHORTESTPATH clause of Cypher query cannot be applied here for the present study due to the following reasons:

1. The SHORTESTPATH function finds the shortest path based on the number of relationships without taking into consideration relationship properties.
2. The SHORTESTPATH clause requires a pattern containing a single relationship, while each path within the emergency routes query of Listing 5.24 contains more than one relationship.

Finally, the graphical representation gained by Neo4j database has been sketched against the floor layout, to clarify the acquired emergency routes, and to show the strength of such approach in retrieving indoor navigation paths in general and emergency routes in particular. The graphical drawing is demonstrated in Figure 5.20 below.

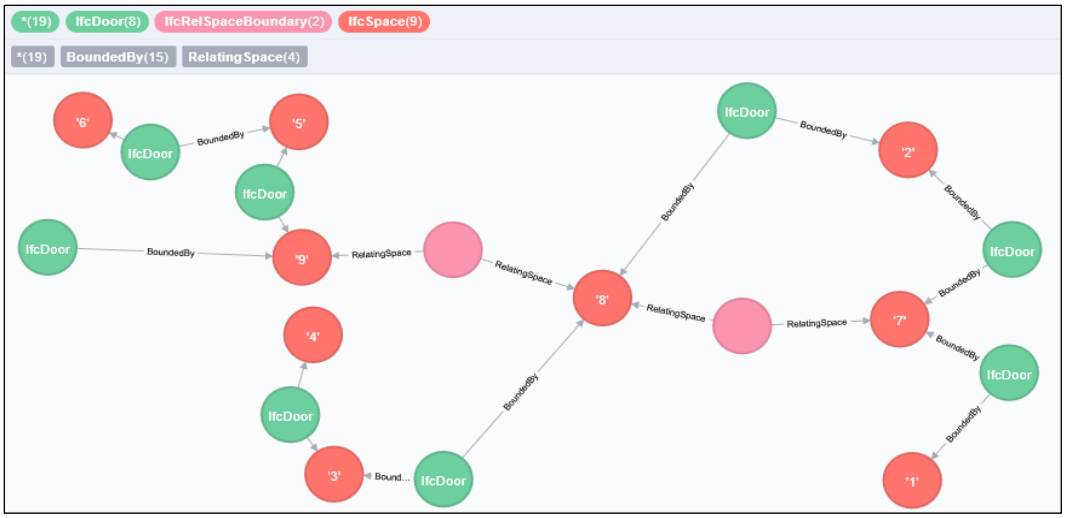

Figure 5.19: Graphical representation of entire floor emergency routes

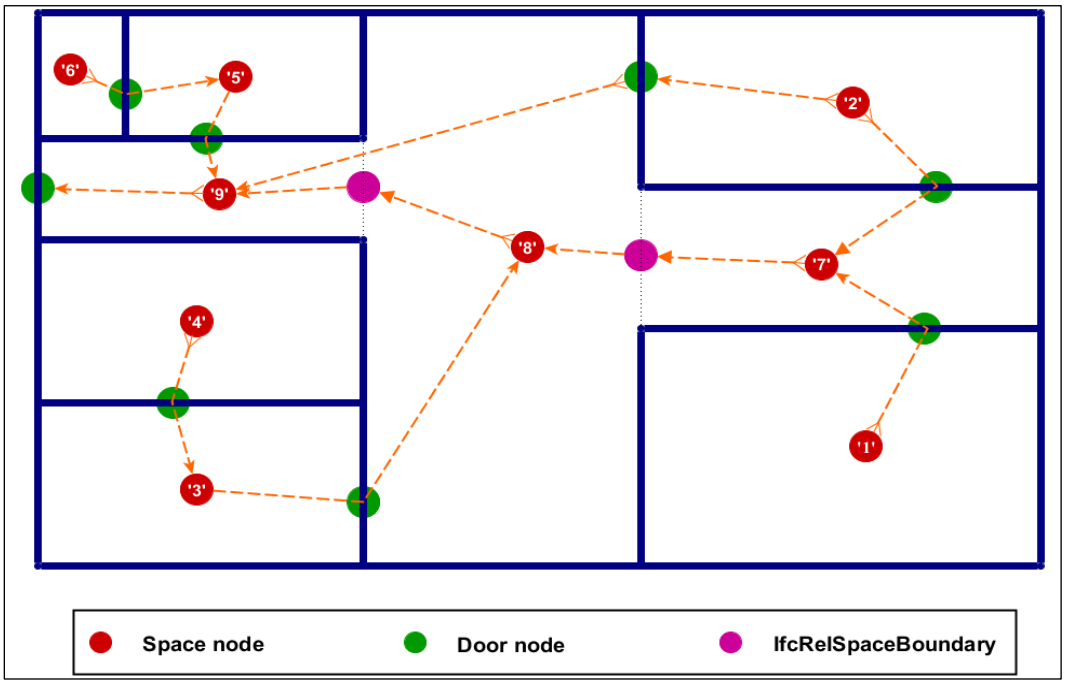

Figure 5.20: Graphical drawing of entire floor emergency routes

To verify the effectiveness of the retrieval query for emergency indoor routes, the above Cypher query of Listing 5.24, will be applied in a more complex model which is the second case study of the present study. The emergency routes query will be applied in the first-floor of the 'Office_A' model, this floor contains 60 spaces as it is shown in the in the 3D Visualization of Figure 5.21 below.

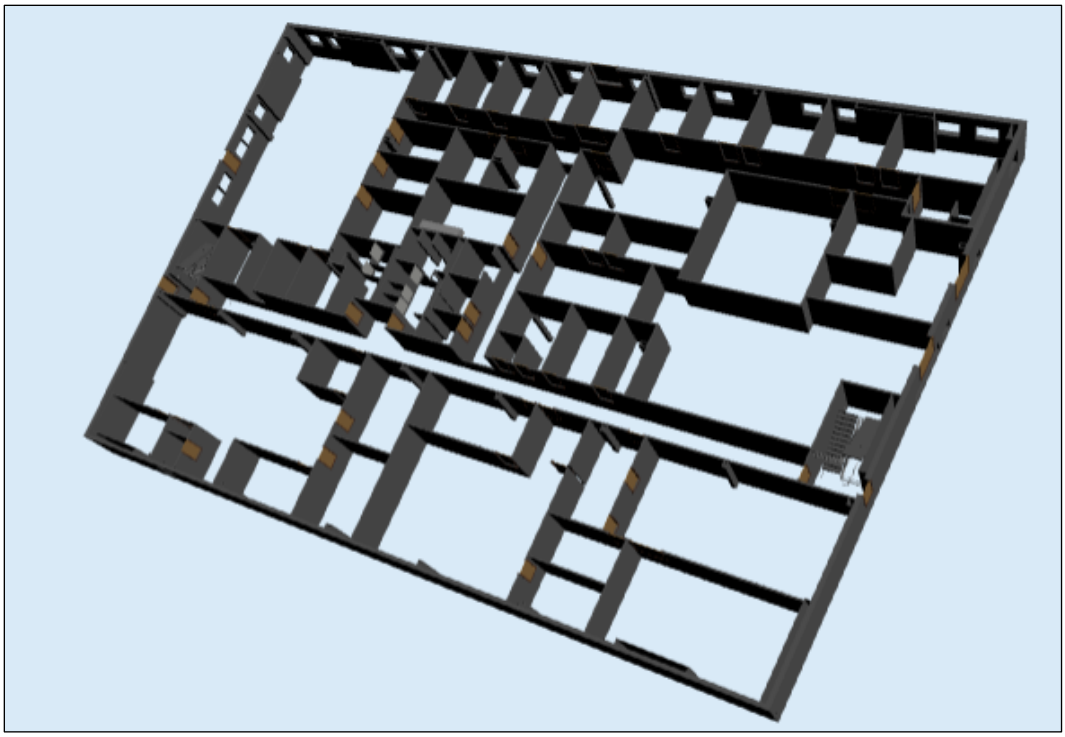

Figure 5.21: 3D Visualization of the first floor of ‘Office_A’ model

To demonstrate the indoor emergency routes within this floor, it is important to illustrate the layout of this floor where the possible access and emergency doors can be shown clearly as depicted in Figure 5.22 below. Four exit doors are available to serve special spaces on the floor or general spaces. The exit doors are graphically represented as yellow nodes while the internal access and doors are represented as green nodes. Since that the building contains multiple floors, therefore, the routes query of Listing 5.24 should be enhanced by specifying the floor understudy and linked the search traversal to the 60 spaces within the floor as it is shown in Listing 5.25 below.

```
MATCH (space: IfcSpace)-[]-(storey: IfcBuildingStorey{IFCID:'1116'})
MATCH p=(door:IfcDoor {IFCID:'807', Model:'Office_A.ifc'})-[rel:BoundedBy]-
>( space)
RETURN p
UNION
MATCH (space1: IfcSpace)-[]-(storey: IfcBuildingStorey{IFCID:'1116'})-[]-
(space2: IfcSpace)
MATCH p=( space1{Model:'Office_A.ifc'})<-[rel1: BoundedBy]-(door: IfcDoor)-
[rel2: BoundedBy]->( space2)
WHERE space1.IFCID > space2.IFCID
RETURN p
UNION
MATCH (space1:IfcSpace)-[]-(storey:IfcBuildingStorey{IFCID:'1116'})-[]-
(space2:IfcSpace)
MATCH p=( space1{Model:'Office_A.ifc'})<-[rel1: RelatingSpace]-
(relSpace:IfcRelSpaceBoundary)-[rel2: RelatingSpace]->( space2)
WHERE space1.IFCID > space2.IFCID
RETURN p
```

Listing 5.25: Cypher query to retrieve emergency routes for the first floor of Office_A

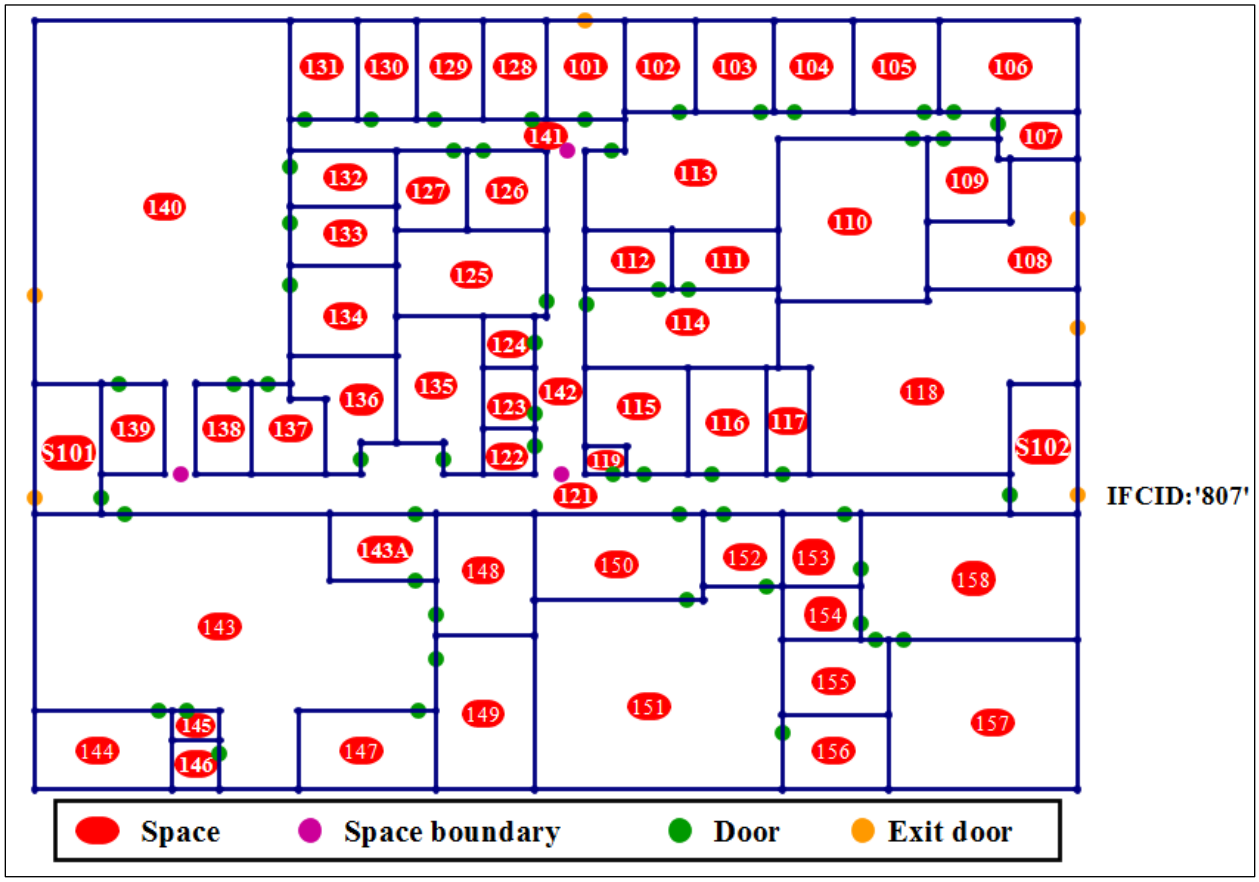

Figure 5.22: First floor's layout of 'Office_A' model

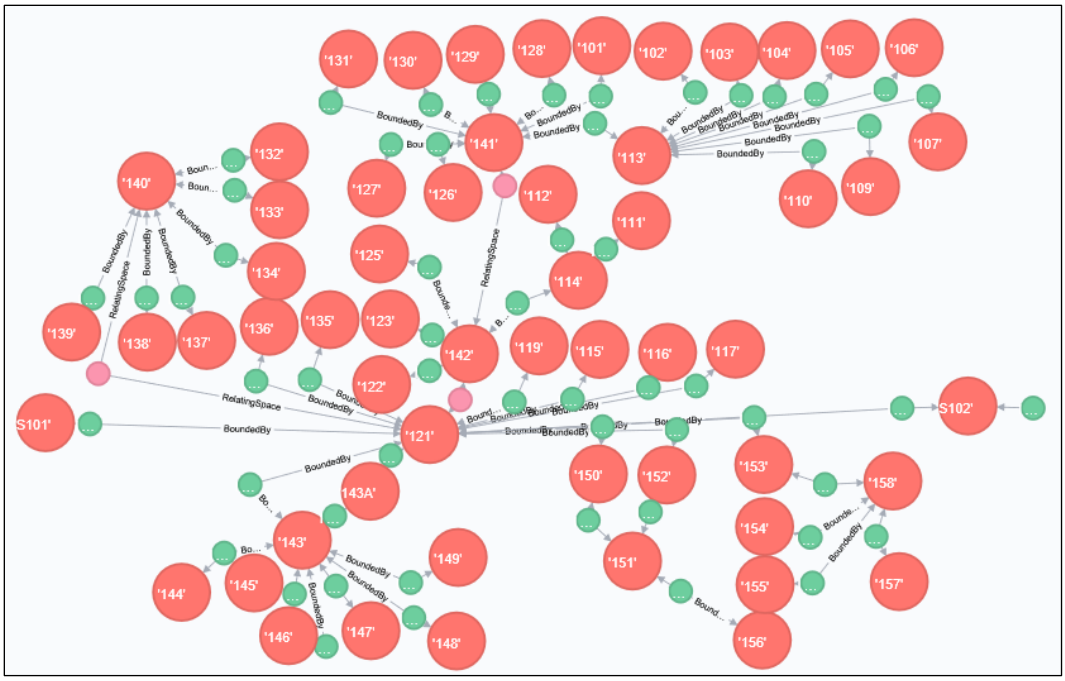

Figure 5.23: Graphical representation of emergency routes using single exit door

The outcome of the graph database has been graphically clarified on the first-floor layout as shown in Figure 5.24 below.

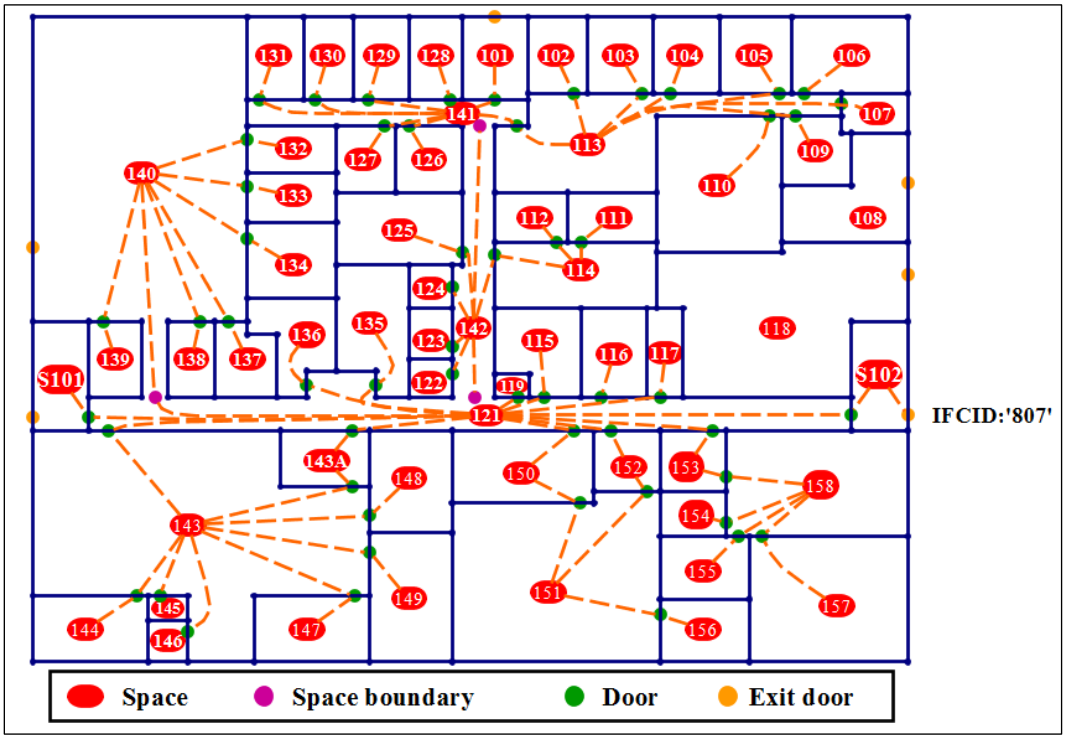

Figure 5.24: Graphical drawing of emergency routes using single exit door

Chapter 6: Discussion and conclusion

Through the filters and queries applications in the present study, it is possible to observe that graph database can effectively be used to explore and analyze BIMs. Thus, graph diagrams tend to represent the data extracted from the IFC models file in form of nodes and directional relationships, this allow users to run complex queries that cannot be run using typical BIM platforms, in addition to the fact that fewer details are required to describe the requested information while applying filters and queries. From the above it is clear that the following objectives have been accomplished:

1. The IMG model that been developed in Chapter 3 of the present study, has provided an interactive database system to analyze the complex data connectivity within the IFC schema, which can sufficiently support understanding of IFC data schema architecture.
2. Various IFC schema released at different period can easily be stored using a single database, to perform a comparison study with aim trace new features. Since that the IMG models have shown high accuracy during the data profiling processes to uncover variations and differences between the original IFC schema data, and the datasets that developed in the graph database.
3. The query examples that been executed on the IOG models demonstrate the flexible relationships that offered by graph networks to provide realistic answers for typical queries. On the other side, it expresses the capability of the IOG models in exploring some unseen relationships inside BIMs. These query examples can be considered as pre-defined graph-based queries for users with basic programming skills to run queries that match their studies based on the examples shown in Chapter 5.
4. Graphical representation for the possible emergency indoor routes has been acquired based on geometric and spatial information extracted from the IFC model, as an example for the utilization of graph theory for advanced analysis of BIMs.

The automation methodology that proposed in this study for graph modeling can be considered as an initial step in the direction of IFC-based graph models formation. Thus, several limitations have been identified in this study, which can affect the reliability of graphs database for use as a mechanism for querying building information representation, and using of graph concepts to visualize and analyze BIMs. First, the study did not provide any techniques for data loss prevention during the data transmission process. Nevertheless, it provides a methodology to examine the correctness of the generated graph database at any stage during the data transmission process. Second, the data profiling process itself need to improve the relationships verification process, since the present work did not introduce any procedure regarding the exact number of relationships that need to be verified for each model. Third, a graph representation of the entire IMG or IOG model at one time, is not effective in the case of rich datasets, since that hundreds of nodes and relationships should be presented within the relatively small stream, resulting in overlap of nodes and information.

In conclusion remains to say that the future work is to make use of the adaptability quality that provided by the graph database to improve the quality of BIMs. In which, incorrect or uncompleted building information could be modified within the graph database. Thus, new labels, new attributes and even new relationships can be added to an existing structure without changing the core arrangement of the building information. Then, the modified data are exported from the graph database and resorted to enhance the original IFC model. Future research could involve the addition of properties to relationships such as cardinality, which indicates the number of objects in each of the entities at either end of the relationship, or even inverse relationship where more queries can be run based on the new relationships.

Graph models can be considered a satisfactory tool to represent building information and rich data such as BIMs. However, developing pre-defined graph-based queries for users without programming skills can be quite challenging. Advanced queries should be wrriten by IFC and graph DB experts.

References

[1] S. Isaac, F. Sadeghpour and R. Navon, "Analyzing Building Information using Graph Theory," in *International Association for Automation and Robotics in Construction (IAARC)- 30th ISARC*, Montreal, 2013.

[2] buildingSMART, "www.buildingsmart-tech.org," buildingSMART International Ltd, [Online]. Available: http://www.buildingsmart-tech.org/ifc/IFC4/Add2/html/. [Accessed 27 September 2016].

[3] F. Chionna , F. Argese, V. Palmieri, I. Spada and L. Colizzi , "Integrating Building Information Modeling and Augmented Reality to Improve Investigation of Historical Buildings," *Conservation Science in Cultural Heritage*, vol. Vo15 (2015), no. ISSN 1973-9494, p. 133, 2015.

[4] "IFC Engine DLL," RDF Ltd, [Online]. Available: http://www.ifcbrowser.com. [Accessed 2016 October 15].

[5] C. Fu, G. Aouad, . A. Lee, A. Mashall-Ponting and S. Wu, "IFC model viewer to support nD model application," *Elsevier*, vol. 15, no. 2, p. 178–185, March 2006.

[6] H. Li, H. Liu, . Y. Liu and Y. Wang, "An Object-relational IFC Storage Model based on Oracle database," in *The International Archives of the Photogrammetry- Remote Sensing and Spatial Information Sciences-XXIII ISPRS Congress*, Prague, 2016.

[7] J. L. Hughes, ECE 3020 Mathematical Foundations of Computer Engineering, Atlanta: Lecture notes from Georgia Institute of Technology, 2016.

[8] R. Wilson, Introduction to Graph Theory, Fourth edition ed., Harlow- England: Longman Group Ltd, 1996.

[9] J. Bondy and U. Murty, Graph Theory with Applications- Fifth printing, New York: Elsevier Science Publishing Co., Inc., 1982.

[10] N. Biggs, L. E. Keith and R. Wilson, Graph Theory 1736-1936, New York : Oxford University Press, 1986.

[11] F. Harary, Graph Theory, London: Addison-Wesley Publishing Company, 1969.

[12] A. Cayley, "On the theory of the analytical forms called trees," *Philosophical Magazine Series 4*, vol. 4, no. 1941-5982, pp. 172-176, 2009.

[13] W. Tutte, Graph Theory, Cambridge: Cambridge University Press, 2001.

[14] V. Klee, "Some Unsolved Problems in Plane Geometry," *Mathematics Magazine-Mathematical Association of America,* vol. 52, no. 3, pp. 131-145, May 1979.

[15] G. Battista, I. Tollis, P. Eades and R. Tamassia, Graph Drawing: Algorithms for the Visualization of Graphs, new Jersey: Alan Api, 1999.

[16] K. Ruohonen, "Graph Theory- Finnish lecture notes for the TUT course (MAT-41196)," Tampere University of Technology, Tampere, 2006.

[17] I. Robinson, J. Webber and . E. Eifrem, Graph Databases, Sebastopol: O'Reilly Media, 2015 "Second Edition".

[18] G. Battista, P. Eades, R. Tamassia and I. Tollis, "Algorithms for drawing graphs: An annotated bibliography," *Elsevier B.V.*, vol. 4, no. 5, pp. 235-282, October 1994.

[19] N. Doekemeijer and A. Varbanescu, "A Survey of Parallel Graph Processing Frameworks," Parallel and Distributed Systems Group- Delft University of Technology, Delft, 2014- PDS-2014-003.

[20] R. Angles and C. Gutierrez, "Survey of Graph Database Models," *ACM Computing Surveys*, vol. 40, no. 1, February 2008 .

[21] N. Technology, "Neo4j," Neo Technology, [Online]. Available: https://neo4j.com. [Accessed 30 November 2016].

[22] M. Weise, T. Liebich and J. Wix , "Integrating use case definitions for IFC developments," in *eWork and eBusiness in Architecture, Engineering and Construction*, *ECPPM*, Nice, 2008.

[23] P.-H. Chena, L. Cuia, C. Wana, Q. Yangb, S. K. Tinga and R. Tiong, "Implementation of IFC-based web server for collaborative building design between architects and structural engineers," *Elsevier B.V,* vol. 14, no. 1, pp. 115 -128, October 2004.

[24] J. Zimdars, "Ruby- A programmer's best friend," Ruby Visual Identity Team, 2006. [Online]. Available: https://www.ruby-lang.org. [Accessed 27 November 2016].

[25] "IFCwebserver," 25 January 2016. [Online]. Available: http://ifcwebserver.org/login.rb?q=login. [Accessed 15 October 2016].

[26] Y. Shafranovich, "Common Format and MIME Type for Comma-Separated Values (CSV) Files," SolidMatrix Technologies , Inc., 2005.

[27] W. Lyon, "LazyWebCypher," GitHub Inc., [Online]. Available: http://www.lyonwj.com. [Accessed 08 December 2016].

[28] C. Eastman , J.-m. Lee, Y.-s. Jeong and J.-k. Lee, "Automatic rule-based checking of building designs," *Elsevier B.V.,* p. 1011–1033, 2009.

[29] Y. S. Sohn and S. D. Choi, "System for managing ifc version synchronized with bim and method for managing ifc version thereof," 31 December 2015. [Online]. Available: http://www.freepatentsonline.com/20150379063.pdf. [Accessed 27 November 2016].

[30] B. Vladimir , "IFC BIM-Based Methodology for Semi-Automated Building Energy Performance Simulation," eScholarship- University of California, Berkeley, 2008.

[31] T.-A. Teo and . K.-H. Cho, "BIM-oriented indoor network model for indoor and outdoor combined route planning," *Elsevier Ltd,* vol. 30, no. 3, pp. 268 - 282, 2016.

[32] Y.-H. Lin, Y.-S. Liu, G. Gao, X.-G. Han , C.-Y. Lai and M. Gu, "The IFC-based path planning for 3D indoor spaces," *Elsevier Ltd.,* vol. 27, no. 2, pp. 189-205, April 2013.

[33] N. Skandhakumar, F. Salim, J. Reid, R. Drogemuller and E. Dawson, "Graph theory based representation of building information models for access control applications," *Elsevier B.V.,* vol. 68, p. 44–51, August 2016.

[34] M. Laakso and A. Kiviniemi, "The IFC standard- A review of history, development ,and standardization," *Information Technology in Construction,* vol. 17, no. 1874-4753, pp. 134-161, May 2012.

[35] L. Mahdjoubi, C. Brebbia and R. Laing, Building Information Modelling (BIM) in Design, Construction and Operations, Southampton: WIT Press, 2015.

[36] X. Kong and P. S. Yu, "Multi-label Feature Selection for Graph Classification," in *2010 IEEE International Conference on Data Mining*, Sydney , 13-17 December 2010.

Appendix

A.1 Ruby script to generate the IFC2X3-Meta Graph (IMG) Model (EXPRESS → Cypher)

```
# To run the script: login to http://ifcwebserver.org/script_catalog.rb
# username: ahmednahar, password: time
load_schema_and_extensions
require $ifc_path + '/IfcTypeClasses_IFC2X3_TC1.rb'
$simple_attributes = IfcTypeClasses.constants.map {|c| c.to_s if
IfcTypeClasses.const_get(c).is_a? Class}
def create_class_property_nodes(classname)
  class_vars= {}
  class_obj = Kernel.const_get(classname.to_s.upcase.strip)
  return if not  defined? class_obj::ClassVars
  class_vars = class_obj::ClassVars if  defined? class_obj::ClassVars
  return if class_obj::ClassVars.keys.size == 0
  puts classname + ":</br>"
  puts class_vars
  class_vars.each { |k,v|
  att_name= v.sub("OPTIONAL ","")
  optional= "false"
  optional = "true" if v.include?("OPTIONAL")
  if $ifcClassesNames.values.include?(att_name)
      $meta_file_commands.puts "MATCH (a:#{classname})   MERGE (a)-[:has_property]->
(n:#{k} {class: \"#{att_name}\", optional: #{optional}});\n"
  elsif $simple_attributes.include?(v.sub("OPTIONAL ",""))
      $meta_file_commands.puts "MATCH (a:#{classname})  MERGE (a)-[:has_property]->
(n:#{k} {class: \"#{v.sub("OPTIONAL ","")}\", optional: #{optional}}) ;\n"
  end
  }
    puts "<hr>"
end
$meta_file_commands = File.new($ifc_path +
"/temp/#{$username}/meta_model_commands.txt", 'w')
$ifcClassesNames.values.sort.each { |classname|
next if classname == "Ifc"
$meta_file_commands.puts "CREATE (n:#{classname}) return n;\n"
}
done={}
$ifcClassesNames.values.each { |classname|
class_vars= {}
class_obj = Kernel.const_get(classname.to_s.upcase.strip)
report_columns = ""
last_class= classname
class_obj.ancestors.each { |cl_parent|
  next if cl_parent == IFC
  if last_class.to_s != "" and $ifcClassesNames[cl_parent.to_s].to_s != "" and
last_class.to_s != $ifcClassesNames[cl_parent.to_s].to_s and done[last_class.to_s +
"_" + $ifcClassesNames[cl_parent.to_s].to_s] == nil
      done[ last_class.to_s + "_" + $ifcClassesNames[cl_parent.to_s].to_s]=true
      $meta_file_commands.puts
"MATCH(a:#{last_class}),(b:#{$ifcClassesNames[cl_parent.to_s]}) CREATE (a)-
[r:SUBTYPE_OF]->(b) return r;\n\n"
  end
  last_class = $ifcClassesNames[cl_parent.to_s]
  report_columns += $report_cols_by_class[cl_parent.to_s.upcase.strip] if
$report_cols_by_class[cl_parent.to_s.upcase.strip]
  if  defined? cl_parent::ClassVars
      class_vars = class_vars.merge(cl_parent::ClassVars)
   end
}
```

```
columns = []
report_columns.split('|').each do |i|
                next if i == ""
    if i.split(':').size > 1
                xx=i.split(':')[0]
                i=i.split(':')[1]
                else
                                xx=i
                end
                columns << xx.strip + ":" + "line." + xx.strip
end
}
$meta_file_commands.puts "//Start with Class --> Property"
$ifcClassesNames.values.sort.each { |classname|
next if classname == "Ifc"
create_class_property_nodes(classname)
```

A.2 Cypher script to create IMG database (output of Ruby script A.1)

```
CREATE (n:Ifc2DCompositeCurve) return n;
CREATE (n:IfcActionRequest) return n;
CREATE (n:IfcActor) return n;
CREATE (n:IfcActorRole) return n;
CREATE (n:IfcActuatorType) return n;
CREATE (n:IfcAddress) return n;
CREATE (n:IfcAirTerminalBoxType) return n;
CREATE (n:IfcAirTerminalType) return n;
CREATE (n:IfcAirToAirHeatRecoveryType) return n;
CREATE (n:IfcAlarmType) return n;
CREATE (n:IfcAngularDimension) return n;
CREATE (n:IfcAnnotation) return n;
CREATE (n:IfcAnnotationCurveOccurrence) return n;
CREATE (n:IfcAnnotationFillArea) return n;
CREATE (n:IfcAnnotationFillAreaOccurrence) return n;
CREATE (n:IfcAnnotationOccurrence) return n;
CREATE (n:IfcAnnotationSurface) return n;
CREATE (n:IfcAnnotationSurfaceOccurrence) return n;
CREATE (n:IfcAnnotationSymbolOccurrence) return n;
CREATE (n:IfcAnnotationTextOccurrence) return n;
CREATE (n:IfcApplication) return n;
CREATE (n:IfcAppliedValue) return n;
CREATE (n:IfcAppliedValueRelationship) return n;
.
.
.
.
.....
MATCH(a:IfcCompositeCurve),(b:IfcBoundedCurve)CREATE(a)-[r:SUBTYPE_OF]->(b)return r;
MATCH(a:IfcBoundedCurve),(b:IfcCurve) CREATE (a)-[r:SUBTYPE_OF]->(b) return r;
MATCH(a:IfcActionRequest),(b:IfcControl) CREATE (a)-[r:SUBTYPE_OF]->(b) return r;
MATCH(a:IfcControl),(b:IfcObject) CREATE (a)-[r:SUBTYPE_OF]->(b) return r;
MATCH(a:IfcObject),(b:IfcObjectDefinition) CREATE (a)-[r:SUBTYPE_OF]->(b) return r;
MATCH(a:IfcObjectDefinition),(b:IfcRoot) CREATE (a)-[r:SUBTYPE_OF]->(b) return r;
.
.
.
.
.....
```

A.3 Ruby script to generate IFC Muster003 Object Graph (IOG) model (IFC → Cypher)

```
# To run the script: login to http://ifcwebserver.org/script_catalog.rb
# username: ahmednahar, password: time

load_schema_and_extensions
require $ifc_path + '/IfcTypeClasses_IFC2x3_TC1.rb'
#Settings of this script
$bim = "Muster003.ifc"

use $bim

input_filenames = []
ignore_classes= ["IFC","IfcGeometricSet",
"IfcCompositeCurve",
"IfcArbitraryClosedProfileDef","IfcCurveStyle",
"IfcCircleProfileDef","IfcArbitraryOpenProfileDef","IfcDraughtingPreDefinedCurveFont
",
"IfcProfileDef","IfcAsymmetricIShapeProfileDef","IfcFaceBasedSurfaceModel",
"IfcFaceBound","IfcRectangleProfileDef","IfcIShapeProfileDef","IfcShapeRepresentatio
n","IfcFillAreaStyle",
"IfcCartesianPoint","IfcLocalPlacement","IfcDirection","IfcSurfaceOfLinearExtrusion"
,"IfcFillAreaStyleHatching",
"IfcPlane","IfcCurveBoundedPlane","IfcCompositeCurveSegment","IfcTrimmedCurve","IfcC
ircle","IfcPolyLoop","IfcColourRgb",
"IfcStyledItem","IfcFaceOuterBound","IfcFace","IfcFacetedBrep","IfcConnectedFaceSet"
,"IfcClosedShell","IfcSurfaceStyleRendering",
"IfcOpenShell","IfcExtrudedAreaSolid","IfcCartesianTransformationOperator3D","IfcAxi
s2Placement3D","IfcAxis2Placement2D","IfcPolyline"]

ignore_classes= ["IFC"]
$export_csv_file = File.new($ifc_path +
"/temp/#{$username}/#{$bim}/#{$bim.sub(".ifc","")}_import_commands.txt", 'w')
$export_csv_file.puts "// BIM Model: #{$bim }</br>"

classes_list=Server.classes_list($bim)
classes_list.each { |classname,count_of_objects|
class_vars= {}
next if ignore_classes.include?(classname)
class_obj = Kernel.const_get(classname.to_s.upcase.strip)
$output_format = "to_csv"
report_columns = "Model:'#{$bim}'|label:'#{classname}'|IFCID:line_id"
class_obj.ancestors.reverse.each { |cl_parent|
    report_columns += $report_cols_by_class[cl_parent.to_s.upcase.strip] if
$report_cols_by_class[cl_parent.to_s.upcase.strip]
  if  defined? cl_parent::ClassVars
      class_vars = class_vars.merge(cl_parent::ClassVars)
   end
}

$export_csv_file.puts "load csv with headers from
'#{$server_ip}/temp/#{$username}/#{$bim}/#{classname}.xls' as line FIELDTERMINATOR '
'
create (u:#{classname} {"

puts "load csv with headers from
'#{$server_ip}/temp/#{$username}/#{$bim}/#{classname}.xls' as line FIELDTERMINATOR '
'</br>
create (u:#{classname} {</br>"

columns = []
report_columns.split('|').each do |i|
```

```
    next if i == ""
    if i.split(':').size > 1
        xx=i.split(':')[0]
        i=i.split(':')[1]
    else
        xx=i
    end
    columns << xx.strip + ":" + "line." + xx.strip
end
$export_csv_file.puts  columns.join(",\n")
$export_csv_file.puts "});\n\n\n"

puts  columns.join(",\n")
puts "});


# Start the CSV export
$old_stdout = $stdout.dup
$stdout = File.new($ifc_path + "/temp/#{$username}/#{$bim}/#{classname}.xls", 'w')
class_obj.list report_columns
input_filenames << "#{classname}.xls"
$stdout = $old_stdout
$row_ID = nil
}


simple_attributes = ["BOOLEAN","INTEGER"]
simple_attributes = IfcTypeClasses.constants.select {|c| c.to_s if
IfcTypeClasses.const_get(c).is_a? Class}
simple_attributes <<
["IfcDateTimeSelect","IfcActorSelect","IfcAppliedValueSelect","IfcConditionCriterion
Select","IfcClassificationNotationSelect","IfcCsgSelect","IfcCurveFontOrScaledCurveF
ontSelect","IfcSizeSelect","IfcCurveStyleFontSelect","IfcDefinedSymbolSelect","IfcFi
llStyleSelect","IfcHatchLineDistanceSelect","IfcFillAreaStyleTileShapeSelect","IfcGe
ometricSetSelect","IfcLightDistributionDataSourceSelect","IfcMetricValueSelect","Ifc
PresentationStyleSelect","IfcObjectReferenceSelect","IfcDocumentSelect","IfcLibraryS
elect","IfcMaterialSelect","IfcOrientationSelect","IfcSurfaceStyleElementSelect","If
cSpecularHighlightSelect","IfcSymbolStyleSelect","IfcCharacterStyleSelect","IfcTextS
tyleSelect","IfcTextFontSelect"]

$relation_ID= 0

relationtemplate= "MATCH (n:XXRelationClass {Model:\"#{$bim}\"})
UNWIND split( replace( replace(n.xxRelationAtt,\"(\",\"\"),\")\",\"\"),\",\") as o
MERGE (XXXRelationAtt {Model:\"#{$bim}\", IFCID: replace(o,\"#\",\"\")})
MERGE (p {Model:\"#{$bim}\" ,IFCID: replace( n.XXRelationTargetAtt,\"#\",\"\")})
MERGE (XXXRelationAtt)-[XXRelationName]->(p);\n\n"

puts "//<b>Start with Relations: </b>"

$relation_script_file = File.new($ifc_path +
"/temp/#{$username}/#{$bim}/#{$bim.sub(".ifc","")}_create_relations_commands.txt",
'w')

relations_data = {
 "IfcRelAggregates" =>  { "relatedAtt" => "relatedObjects", "relatingAtt" =>
"relatingObject", "relationName" => ":Decomposes"},
 "IfcRelAssignsTasks" =>  { "relatedAtt" => "relatedObjects", "relatingAtt" =>
"relatingControl", "relationName" => ":HasAssignments"},
 "IfcRelAssignsToGroup" =>  { "relatedAtt" => "relatedObjects", "relatingAtt" =>
"relatingGroup", "relationName" => ":HasAssignments"},
 "IfcRelAssignsToActor" => { "relatedAtt" => "relatedObjects", "relatingAtt" =>
"relatingActor", "relationName" => ":HasAssignments"},
```

```
 "IfcRelAssignsToProcess" =>  { "relatedAtt" => "relatedObjects", "relatingAtt" =>
"relatingProcess", "relationName" => ":HasAssignments"},
 "IfcRelAssociatesClassification" =>  { "relatedAtt" => "relatedObjects",
"relatingAtt" => "relatingClassification" , "relationName" => ":HasAssociations"},
 "IfcRelAssociatesMaterial"  =>  { "relatedAtt" => "relatedObjects", "relatingAtt"
=> "relatingMaterial" ,"relationName" => ":HasAssociations"},
 "IfcRelSpaceBoundary" =>  { "relatedAtt" => "relatedBuildingElement", "relatingAtt"
=> "relatingSpace",  "relationName" => ":BoundedBy" },
 "IfcRelDefinesByProperties" =>  { "relatedAtt" => "relatedObjects", "relatingAtt"
=> "relatingPropertyDefinition",  "relationName" => ":IsDefinedByProperties"},
 "IfcRelDefinesByType" =>  { "relatedAtt" => "relatedObjects", "relatingAtt" =>
"relatingType",  "relationName" => ":IsDefinedByType"},
 "IfcRelFillsElement" =>  { "relatedAtt" => "relatedBuildingElement", "relatingAtt"
=> "relatingOpeningElement",  "relationName" => ":FillsVoids"},
 "IfcRelVoidsElement" =>  { "relatedAtt" => "relatedOpeningElement", "relatingAtt"
=> "relatingBuildingElement",  "relationName" => ":VoidsElements"},
 "IfcRelCoversSpaces" =>  { "relatedAtt" => "relatedCoverings", "relatingAtt" =>
"relatedSpace",  "relationName" => ":HasCoverings"},
 "IfcRelConnectsElements" =>  { "relatedAtt" => "relatedElement", "relatingAtt" =>
"relatingElement",  "relationName" => ":ConnectedFrom"},
 "IfcRelConnectsPathElements" => { "relatedAtt" => "relatedElement", "relatingAtt"
=> "relatingElement",  "relationName" => ":RelConnectsPathElements"},
 "IfcRelConnectsPorts" =>  { "relatedAtt" => "relatedPort", "relatingAtt" =>
"relatingPort",  "relationName" => ":RelConnectsPorts"},
 "IfcRelConnectsPortToElement" =>  { "relatedAtt" => "relatedElement", "relatingAtt"
=> "relatingPort",  "relationName" => ":RelConnectsPortToElement"},
 "IfcRelContainedInSpatialStructure" =>  { "relatedAtt" => "relatedElements ",
"relatingAtt" => "relatingStructure",  "relationName" => ":ContainsElements"},
 "IfcRelCoversBldgElements" =>  { "relatedAtt" => "relatedCoverings", "relatingAtt"
=> "relatingBuildingElement",  "relationName" => ":RelCoversBldgElements"},
 "IfcRelNests" =>  { "relatedAtt" => "relatedObjects", "relatingAtt" =>
"relatingObject",  "relationName" => ":RelNests"},
 "IfcRelSequence" =>  { "relatedAtt" => "relatedProcess", "relatingAtt" =>
"relatingProcess",  "relationName" => ":RelSequence"},
 "IfcRelServicesBuildings" =>  { "relatedAtt" => "relatedBuildings", "relatingAtt"
=> "relatingSystem",  "relationName" => ":RelServicesBuildings"}
}

relations_data.each { |rel_class, rel_atts|
command =
relationtemplate.gsub("XXRelationClass",rel_class).gsub("XXXRelationAtt",rel_atts["r
elatedAtt"]).gsub("xxRelationAtt",rel_atts["relatedAtt"]).sub("XXRelationName",rel_a
tts["relationName"]).sub("XXRelationTargetAtt",rel_atts["relatingAtt"])
puts "<div style='background-color: lightgray'><pre><code>" + command +
"</pre></code></div>"
$relation_script_file.puts command
}


classes_list=Server.classes_list($bim)

classes_list.each { |classname,count_of_objects|
next if ignore_classes.include?(classname)
next if classname == "Ifc"

class_vars= {}
class_obj = Kernel.const_get(classname.to_s.upcase.strip)
class_obj.ancestors.reverse.each { |cl_parent|
  class_vars = class_vars.merge(cl_parent::ClassVars) if  defined?
cl_parent::ClassVars
}
class_vars.each { |k,v|
att_name= v.sub("OPTIONAL ","")
```

```
if $ifcClassesNames.values.include?(att_name)
    $relation_script_file.puts  "//" + classname + "." + k.to_s + ": " +  att_name +
    puts  + classname + "." + k.to_s + ": " +  att_name +
    command = "match(o:#{classname}  {Model:\"#{$bim}\"})\n"
    command += "MERGE (att {Model:\"#{$bim}\" , IFCID:
replace(o.#{k},\"#\",\"\")})\n"
    command +=  "MERGE (o)-[:#{k}]->(att);\n\n"
    puts "<div style='background-color: lightgray'><pre><code>" + command +
"</pre></code></div>"
    $relation_script_file.puts command
else
    if not simple_attributes.to_s.include?(att_name.split(" ")[-1].strip.gsub("
","""))
        if att_name.include?("IfcRel") and att_name.include?(" FOR") and not
att_name.include?("IfcRelAssociates")
          xxRelationClass  = "IfcRel" + att_name[/#{"IfcRel"}(.*?)#{" FOR"}/m,
1].gsub(" ","")
          #Igonre the Relationship if we already added it before relations_data
          next if relations_data[xxRelationClass] != nil
          #Ignore the ownerHistroy relationship
          next if k.to_s == "ownerHistory"
          xxRelationAtt = att_name.split(" ")[-1].strip.gsub(" ","").to_s
          xxRelationClass_obj = Kernel.const_get(xxRelationClass.to_s.upcase.strip)
          if xxRelationClass_obj::ABSTRACT
          else
            target_att_name = att_name.split(" ")[-1]
            $report_cols_by_class[xxRelationClass.upcase].sub("|" +
target_att_name[0, 1].downcase + target_att_name[1..-1],"")
            xxRelationTargetAtt = "UNKNOWN"
            if  $report_cols_by_class[xxRelationClass.upcase].split("|").size  == 2
                xxRelationTargetAtt =
$report_cols_by_class[xxRelationClass.upcase].sub("|" ,"")
            else
            end

            #puts "</br>XXRelationTargetAtt:" + xxRelationTargetAtt
            next if ignore_classes.include?(xxRelationClass)
            command =
relationtemplate.gsub("XXRelationClass",xxRelationClass).gsub("XXXRelationAtt",xxRel
ationAtt).gsub("xxRelationAtt", xxRelationAtt[0, 1].downcase + xxRelationAtt[1..-1]
).sub("XXRelationName", ":" + k.to_s).sub("XXRelationTargetAtt",xxRelationTargetAtt)
            puts "<div ><pre><code>" + command + "</pre></code></div>"
          end
        end
        $relation_ID += 1

        if not att_name.split(" ")[-1].to_s.include?("BOOLEAN") and not
att_name.split(" ")[-1].to_s.include?("INTEGER")  and not att_name.split(" ")[-
1].to_s.include?("Enum")
            puts $relation_script_file.puts  "//TODO#{$relation_ID}: " + classname +
"." + k.to_s + ": " +  att_name + "</br>\n"
            puts TODO#{$relation_ID}:" + classname + "." + k.to_s + ": " +  att_name
+ " >>>> #{att_name.split(" ")[-1]}
        end
    end
end
}
}
$relation_script_file.close
```

A.4 Cypher script to create IOG database (output of Ruby script A.3)

```
// BIM Model:  Muster003.ifc
load csv with headers from
'http://www.ifcwebserver.org/temp/ahmednahar/Muster003.ifc/IfcApplication.xls' as
line FIELDTERMINATOR ' '
create (u: IfcApplication {
Model: line.Model, label: line.label, IFCID: line.IFCID, applicationDeveloper:
line.applicationDeveloper, version: line.version, applicationFullName:
line.applicationFullName, applicationIdentifier: line.applicationIdentifier });

load csv with headers from
'http://www.ifcwebserver.org/temp/ahmednahar/Muster003.ifc/IfcArbitraryOpenProfileDe
f.xls' as line FIELDTERMINATOR ' '
create (u: IfcArbitraryOpenProfileDef {
Model: line.Model, label: line.label, IFCID: line.IFCID, profileType:
line.profileType, profileName: line.profileName, curve: line.curve });

load csv with headers from
'http://www.ifcwebserver.org/temp/ahmednahar/Muster003.ifc/IfcAxis2Placement2D.xls'
as line FIELDTERMINATOR '    '
create (u: IfcAxis2Placement2D {
Model: line.Model, label: line.label, IFCID: line.IFCID, location: line.location,
refDirection: line.refDirection });

load csv with headers from
'http://www.ifcwebserver.org/temp/ahmednahar/Muster003.ifc/IfcAxis2Placement3D.xls'
as line FIELDTERMINATOR '    '
create (u: IfcAxis2Placement3D {
Model: line.Model, label: line.label, IFCID: line.IFCID, location: line.location,
axis: line.axis, refDirection: line.refDirection });

load csv with headers from
'http://www.ifcwebserver.org/temp/ahmednahar/Muster003.ifc/IfcBuilding.xls' as line
FIELDTERMINATOR '    '
create (u: IfcBuilding {
Model: line.Model, label: line.label, IFCID: line.IFCID, globalId: line.globalId,
ownerHistory: line.ownerHistory, name: line.name, description: line.description,
objectType: line.objectType, objectPlacement: line.objectPlacement, representation:
line.representation, longName: line.longName, compositionType: line.compositionType,
elevationOfRefHeight: line.elevationOfRefHeight, elevationOfTerrain:
line.elevationOfTerrain, buildingAddress: line.buildingAddress });

.
.
.
.
.
.

load csv with headers from
'http://www.ifcwebserver.org/temp/ahmednahar/Muster003.ifc/IfcWallType.xls' as line
FIELDTERMINATOR '    '
create (u: IfcWallType {
Model: line.Model, label: line.label, IFCID: line.IFCID, globalId: line.globalId,
ownerHistory: line.ownerHistory, name: line.name, description: line.description,
applicableOccurrence: line.applicableOccurrence, hasPropertySets:
line.hasPropertySets, representationMaps: line.representationMaps, tag: line.tag,
elementType: line.elementType, predefinedType: line.predefinedType });
```

A.5 Cypher script to add multi-labels to IOG models in Neo4j

```
match (n:Ifc2DCompositeCurve) set
n:IfcCompositeCurve:IfcBoundedCurve:IfcCurve:IfcGeometricRepresentationItem:IfcRepre
sentationItem;
match (n:IfcActionRequest) set n:IfcControl:IfcObject:IfcObjectDefinition:IfcRoot;
match (n:IfcActor) set n:IfcObject:IfcObjectDefinition:IfcRoot;
match (n:IfcActuatorType) set
n:IfcDistributionControlElementType:IfcDistributionElementType:IfcElementType:IfcTyp
eProduct:IfcTypeObject:IfcObjectDefinition:IfcRoot;
.
.
.
.
.
.
.
.
match (n:IfcWindow) set
n:IfcBuildingElement:IfcElement:IfcProduct:IfcObject:IfcObjectDefinition:IfcRoot;
match (n:IfcWindowLiningProperties) set
n:IfcPropertySetDefinition:IfcPropertyDefinition:IfcRoot;
match (n:IfcWindowPanelProperties) set
n:IfcPropertySetDefinition:IfcPropertyDefinition:IfcRoot;
match (n:IfcWindowStyle) set
n:IfcTypeProduct:IfcTypeObject:IfcObjectDefinition:IfcRoot;
match (n:IfcWorkControl) set n:IfcControl:IfcObject:IfcObjectDefinition:IfcRoot;
match (n:IfcWorkPlan) set
n:IfcWorkControl:IfcControl:IfcObject:IfcObjectDefinition:IfcRoot;
match (n:IfcWorkSchedule) set
n:IfcWorkControl:IfcControl:IfcObject:IfcObjectDefinition:IfcRoot;
match (n:IfcZShapeProfileDef) set n:IfcParameterizedProfileDef:IfcProfileDef;
match (n:IfcZone) set n:IfcGroup:IfcObject:IfcObjectDefinition:IfcRoot;
```