EFFICIENT QUERYING OF SBGN MAPS STORED IN A GRAPH DATABASE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Mustafa Enes Karaca
February 2019

Efficient Querying of SBGN Maps Stored in a Graph Database
By Mustafa Enes Karaca
February 2019

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Uğur Doğrusöz(Advisor)

Ali Aydın Selçuk

Buğra Gedik

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

EFFICIENT QUERYING OF SBGN MAPS STORED IN A GRAPH DATABASE

Mustafa Enes Karaca
M.S. in Computer Engineering
Advisor: Uğur Doğrusöz
February 2019

Graph visualization is an important research area that endeavors to make graphs more understandable and easier to analyze. In various domains, graph visualization techniques and standards are developed to effectively analyze underlying graph based data. Systems Biology Graphical Notation (SBGN) is a standard language for modeling biological processes and pathways through graph visualization. Information about SBGN maps can be stored in XML based SBGN-ML files. libSBGN is a Java/C++ library for reading, writing SBGN-ML and manipulating SBGN maps in an object-oriented manner.

Graph databases store data in terms of a graph structure consisting nodes and their relationships. Performing a computation on graph data stored in a graph database by traversals is more efficient than accessing tabled data in relational databases through costly join operations. Neo4j is a prominent graph database that provides a proprietary language named Cypher for querying stored graph data. Neo4j allows writing user defined procedures in Java as plugins to improve capabilities of Neo4j with third party Java libraries.

With this thesis, we enable modeling SBGN maps in Neo4j graph database with support for compound structures. Using this SBGN data model in Neo4j, we developed graph based user defined procedures in Java using libSBGN as a plugin to Neo4j. These procedures were used to implement graph query algorithms, such as neighborhood, common stream, and paths between, along with helper functions such as populating a database from an SBGN map and loading an SBGN map from a graph database. These user defined procedures are designed to produce or consume SBGN-ML; hence, they can be used by any visualization tool which can import/export SBGN-ML text. Newt, a web based editor for viewing and

editing SBGN maps, is such a tool making use of these procedures and hosting a local Neo4j instance by providing a web service to execute Cypher statements.

Keywords: Graph algorithms, graph visualization, systems biology, graph databases, graph query, SBGN, Neo4j, Cypher, Newt, libSBGN, SBGN-ML.

ÖZET

ÇİZGE VERİ TABANINDA DEPOLANAN SBGN HARİTALARININ ETKİLİ SORGULANMASI

Mustafa Enes Karaca
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Uğur Doğrusöz
Şubat 2019

Çizge görselleştirme, çizgeleri daha anlaşılır ve temsil edilen verilerin analizini daha kolay hale getirmeye çalışan önemli bir araştırma alanıdır. Çeşitli alanlarda, temel çizge tabanlı verilerin etkin bir şekilde analiz edilmesi için çizge görselleştirme teknikleri ve standartları geliştirilmiştir. Systems Biology Graphical Notation (SBGN), biyolojik süreçleri ve yolakları çizge görselleştirme yöntemiyle modellemek için standart bir dildir. SBGN haritaları hakkındaki bilgiler XML tabanlı SBGN-ML dosyalarında saklanabilir. libSBGN, SBGN-ML dosyalarını okumak, yazmak ve nesne yönelimli bir şekilde SBGN haritalarını değiştirmek için kullanılan bir Java/C++ kütüphanesidir.

Çizge veritabanları, verileri düğümlerden (köşe) ve bunların ilişkilerinden (kenar) oluşan bir çizge yapısı olarak saklar. Bir çizge veritabanında çizge dolaşma yöntemiyle depolanan verilere erişmek, maliyetli birleşim işlemleri yoluyla ilişkisel veritabanlarındaki tablolanmış verilere erişmekten daha etkilidir. Neo4j, depolanmış çizge verilerini sorgulamak için Cypher adlı tescilli bir dil sağlayan seçkin bir çizge veritabanıdır. Neo4j, üçüncü taraf Java kütüphaneleriyle Neo4j'nin yeteneklerini geliştirmek için Java'da yazılmış kullanıcı tanımlı prosedürleri eklenti olarak geliştirme olanağı sağlar.

Bu tez ile, Neo4j çizge veritabanında, SBGN haritalarının, bileşik yapılı düğümleri de destekleyerek modellenmesini sağlıyoruz. Neo4j'de oluşturduğumuz SBGN veri modelini ve libSBGN kütüphanesini kullanarak, Java'da Neo4j eklentisi olarak, çizge tabanlı kullanıcı tanımlı prosedürler geliştirdik. Bu prosedürler, çizge sorgulama algoritmaları yanı sıra bir SBGN haritasından bir veritabanını oluşturmak, bir çizge veritabanından bir SBGN haritasını yüklemek gibi yardımcı işlevler sağlamak için geliştirilmiştir. Bu kullanıcı tanımlı prosedürler SBGN-ML üretmek veya tüketmek için tasarlanmıştır; bu nedenle, SBGN-ML metnini

içe/dışa aktarabilen herhangi bir görselleştirme aracı tarafından kullanılabilirler. SBGN haritalarını görüntülemek ve düzenlemek için web tabanlı bir editör olan Newt, bu prosedürleri Cypher ifadeleri ile koşturmak için bir web servisi kullanan ve yerel bir Neo4j veritabanına ev sahipliği yapan bir araç haline getirilmiştir.

Anahtar sözcükler: Çizge algoritmaları, çizge görşelleştirme, sistem biyolojisi, çizge veritabanları, çizge sorgusu, SBGN, Neo4j, Cypher, Newt, libSBGN, SBGN-ML.

Acknowledgement

I would like to express my sincere thankfulness to my supervisor Prof. Uğur Doğrusöz for providing me opportunity to work with him and defining a thesis topic that I studied with pleasure. During my graduate study, his support and guidance always helped me to proceed on my research. He allocated his precious time to review my thesis. I have learned a lot from him.

I would like to thank Prof. Ali Aydın Selçuk and Assoc. Prof. Buğra Gedik for reviewing and commenting on the manuscript of my thesis.

I would like to thank to TUBITAK ILTAREN for supporting my graduate study and for their understanding.

I would like to express my deepest gratitude to my father, my mother and my sister for their priceless support, understanding and patience throughout my life.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Contribution 2

2 Background and Related Work **4**

2.1 Graph Visualization 4

2.2 SBGN 5

2.2.1 Process Description Language 5

2.2.2 libSBGN and SBGN-ML 7

2.3 Neo4j 9

2.3.1 Property Graph Data Model 11

2.3.2 Cypher 12

2.3.3 User-Defined Procedure 12

2.4 Cytoscape 13

2.5 Newt 14

2.6 STON 17

3 Graph Based User Defined Procedures for SBGN Maps 18

3.1 Data Model 18

3.2 Procedures on Cypher for Query Algorithms 21

3.2.1 General Rules and Methods 21

3.2.2 Populating Graph Database from an SBGN Map 25

3.2.3 Loading an SBGN Map from the Database 28

3.2.4 Neighbourhood Procedure 28

3.2.5 Common Stream Procedure 32

3.2.6 Paths-Between Procedure 35

3.2.7 Paths-From-To Procedure 38

3.3 Experiments on Performance 41

4 Adding Local Database Support to Newt 51

4.1 Save to Graph DB 53

4.2 Read From Graph DB 53

4.3 Neighbourhood Query 54

4.4 Common Stream Query 55

4.5 Paths-Between Query 56

4.6 Paths-From-To Query 57

5 Conclusion **59**

5.1 Future Work 60

List of Figures

2.1 Process Description Language Reference Card [1] 6

2.2 Complex containing entity pool nodes 7

2.3 SBGN-ML Sample 8

2.4 Corresponding SBGN diagram of the SBGN-ML sample in Figure 2.3 8

2.5 Neo4j Browser 10

2.6 Property graph data model example for book store [2] 11

2.7 Example Cypher statement for retrieving 1-neighbourhood of a process node. 12

2.8 Workflow diagram for user defined procedures [3] 13

2.9 A biological network visualization by Cytoscape [4] 14

2.10 Architecture of Newt [5] 15

2.11 Sample screenshot from the Newt Editor 16

3.1 Cypher statement to get macromolecules with state-variable P@T287 and state-variable P@S334 21

3.2 Sample result path (highlighted) and the extended network that is intact as taken from Newt 23

3.3 Cypher statement for getting a compound node and its members including parent, child and siblings. 23

3.4 Sample compound graph containing nested complexes and macromolecules 24

3.5 Visualization of the compound graph in Figure 3.4 as represented by Neo4j data model. 24

3.6 Visualization of a sample SBGN map in Newt 26

3.7 How sample SBGN map in Figure 3.6 is rendered by Neo4j browser in line with the data model 27

3.8 Cypher statement for retrieving all nodes from graph db 28

3.9 Cypher statement for retrieving all relationships from graph db . 28

3.10 Sample graph in SBGN 30

3.11 Neighbourhood Query Dialog 31

3.12 Common Stream Query Result - 1 33

3.13 Common Stream Query Result - 2 34

3.14 Paths-Between Query Result - 1 36

3.15 Paths-Between Query Result - 2 37

3.16 Paths-From-To Query Result - 1 39

3.17 Paths-From-To Query Result - 2 40

3.18 Neighbourhood Procedure - Execution Time vs Number of Nodes chart on large SBGN maps. Notice that the number of edges in returned graphs should be proportional to the number of nodes in SBGN maps. 42

3.19 Common Stream Procedure - Execution Time vs Number of Nodes chart on large SBGN maps. Notice that the number of edges in returned graphs should be proportional to the number of nodes in SBGN maps. 43

3.20 Paths-Between Procedure - Execution Time vs Number of Nodes chart on large SBGN maps. Notice that the number of edges in returned graphs should be proportional to the number of nodes in SBGN maps. 44

3.21 Paths-From-To Procedure - Execution Time vs Number of Nodes chart on large SBGN maps. Notice that the number of edges in returned graphs should be proportional to the number of nodes in SBGN maps. 45

3.22 Paths-Between Procedure - Execution Time vs Result Size (number of nodes and edges) chart on SBGN map with 4514 nodes . . 46

3.23 Paths-From-To Procedure - Execution Time vs Length chart on SBGN map with 4514 nodes 47

3.24 Execution Times of procedures on the SBGN map with 1065 nodes vs Length 48

3.25 Execution Times of procedures on the SBGN map with 1281 nodes vs Length 49

3.26 Execution Times of procedures on the SBGN map with 880 nodes vs Length 50

4.1 Architecture of Newt and Neo4j Integration 52

4.2 Menu item to save graph to database 53

4.3 Menu item to read from the graph database 54

4.4 Neighbourhood query dialog 55

4.5 Common Stream query dialog 56

4.6 Paths-Between query dialog 57

4.7 Paths-From-To query dialog 58

List of Tables

3.1 Mapping from a glyph to a node 19

3.2 Mapping from an arc to a relationship 20

Chapter 1

Introduction

Graph is an abstract way to represent relational information with a set of objects called nodes, and relationships among these objects. Graph visualization is a leading research field that strives to make relational data understandable, meaningful and easy to work on in various domains.

System Biological Notation (SBGN), which is designed by a group of computer scientist and biochemists is a notational language for modeling biological processes and pathways through graph visualization [6]. SBGN provides three basic languages which are processes description (PD), activity flow (AF) and entity relationship (ER) [1]. Furthermore, SBGN supports compound structures, which are typically represented with nested sub-graphs in graph visualization.

SBGN-ML is an XML based file format for storing information of SBGN maps and libSBGN is a Java/C++ library for reading, writing and manipulating SBGN maps [7].

Neo4j which is database management system stores relational information in terms of property graph [2]. Furthermore, Neo4j allows writing user defined procedures to improve its capabilities with third party Java libraries [8].

Newt, is a web tool for visualizing and editing SBGN maps has features of

importing and exporting SBGN-ML file [5].

1.1 Motivation

As an SBGN map is a visual graph that contains a set of edges and nodes, modeling such representations in a graph database facilitates easy access and manipulation of SBGN maps. Accessing or querying of graph databases through traversals is a lot more efficient than traversals using join operations in relational, table based databases [9].

Neo4j implements property graph data model, so modeling SBGN in Neo4j can be done by directly mapping glyphs of SBGN to nodes of property graph data model and arcs of SBGN to relationships of property graph data model. Neo4j's support for user defined procedures lets us use libSBGN library with Neo4j database.

As Newt can import and export SBGN-ML text, it is useful to implement user defined procedures that takes or return SBGN-ML text. Consequently, we can provide user defined procedures that takes or return SBGN-ML text as plugin to Neo4j by libSBGN and Neo4j's Java driver in order to run graph algorithms to visualize resulted SBGN graph by Newt.

1.2 Contribution

In this thesis, we model SBGN maps in Neo4j with support for compound structures, using a similar representation to that in [10]. We handle compound structures during modeling by creating a dedicated *resideIn* relationship between parent and child nodes.

In this respect, we developed a compound based breadth first traversal method for proper traversal of SBGN maps in Neo4j respecting compound structures and

bipartite nature of these graphs. At this point, Cypher statements which define patterns with *resideIn* relationship was used for handling compound structures.

By using a compound breadth first traversal method together with libSBGN, graph based user defined procedures were implemented and packaged in Java as a plugin to Neo4j. Procedures were designed to take or return SBGN-ML text. This graph based user defined procedures can be used with any visualization tool that can produce or consume SBGN-ML text and has connection to Neo4j graph database.

Once a graph based user defined procedures were developed and deployed, we were able to integrate Newt with a Neo4j database for executing these graph based user defined procedures through Newt's user friendly interface. In order to call graph based procedures, we provide web services that execute Cypher statements.

Chapter 2

Background and Related Work

2.1 Graph Visualization

A graph G = (V,E) is a set of vertices or nodes V and set of edges E. Two vertices (u,v) are connected by edge (e=(u,v)).

An edge can be directed or undirected. Directed graph means that there is an incoming or outgoing edge from a vertex to another vertex. When an edge can be bi-directed, from vertices (u) to (v).

A vertex can be a compound vertex, which contains other vertices and edges.

A graph which is formed by edges and vertices is an abstract way to represent relational information. In other words, a graph can contain numerous edges and vertices to represent networks formed by relational information of various domains such as biology and social media. In order to make these networks of relational information meaningful and easy to work on, visualization of graph is crucial.

Graph visualization is about drawing graphs for edges and vertices of graph by using geometric shapes and attributes of geometric shapes such as location, width, and height. Moreover, many styling attributes such as color, thickness, contrast

are considered useful aspects of graphical visualization. In fact, any visual feature that users can understand is considered as a graphical visualization concept [11].

2.2 SBGN

The System Biological Graphical Notation (SBGN) is a standard language developed by a group of biochemists, biologist and computer scientists in order to visualize biological processes and interactions in the domain of system biology [6]. SBGN contains three basic and complementary languages. These languages are the process description, entity relationship, and activity flow [6].

2.2.1 Process Description Language

In order to visualize molecular processes and interactions, SBGN process diagrams (PD) are used. PD language shows how molecular entities involve in biological processes and interactions [6]. Process description language illustrates how entities pass from one form to another as a result of different effects [6, 1]. The main disadvantage of the process diagram is that a particular entity may appear more than once in the same diagram if it exists in more than one state. Therefore, it leads to several occurrences of possible entities and reactions in the SBGN map [1].

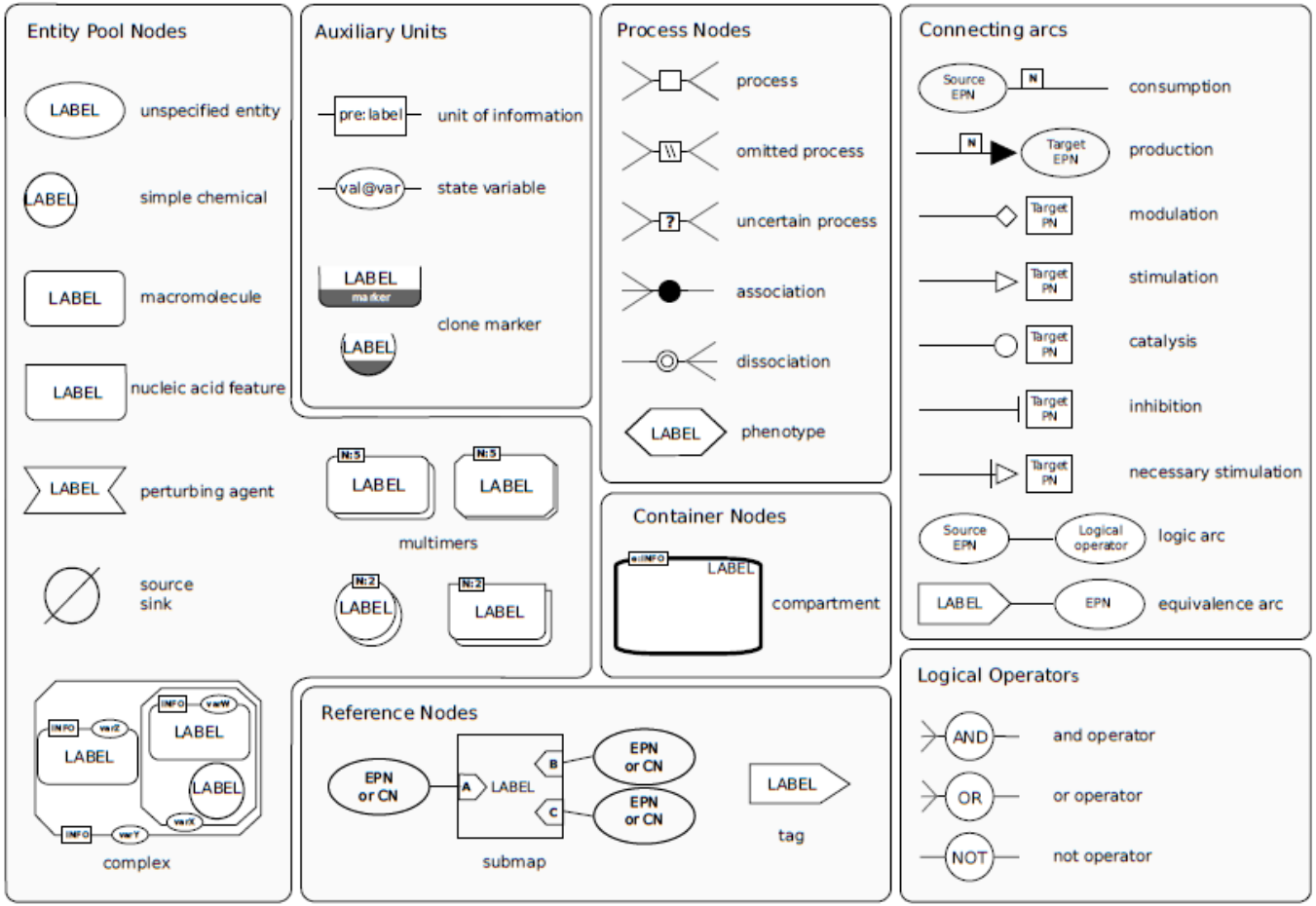


Figure 2.1: Process Description Language Reference Card [1]

PD language contains several types of elements. These elements are defined under headings of entity pool nodes, process nodes, container nodes, reference nodes, logical operators, auxiliary units and connecting arcs [1]. Entity pool nodes consist some type of glyphs: unspecified entity, macromolecule, complex, simple chemical. Unspecified entity is a entity whose type is not known. Simple chemical represents chemical compounds. Macromolecule represents elements that make biological process occurs. Complexes symbolize compound entities that contain other entity pool nodes (Figure 2.2). Process nodes represent the operations to convert one or more entity pools into one or more others. Process nodes can be process, omitted process, uncertain process, association or dissociation [1].

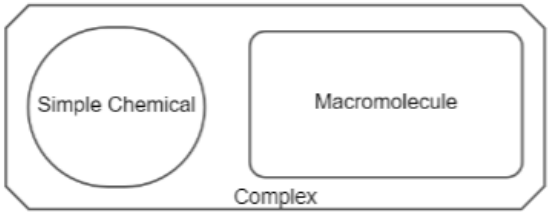


Figure 2.2: Complex containing entity pool nodes

2.2.2 libSBGN and SBGN-ML

libSBGN is an object-oriented software library providing functions for reading, writing and editing SBGN-ML file, an XML-based file format for storing SBGN map that is basically a collection of glyphs and arcs [7]. While the node corresponds to the glyph, the edge corresponds to the arc in SBGN-ML format. Also, the format of SBGN-ML file stores all the required information to let visualization tools to draw SBGN-map without any additional information or calculation (Figure 2.3, Figure 2.4) [7].

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<sbgn xmlns="http://sbgn.org/libsbgn/0.2">
  <map language="unknown">
    <glyph class="compartment" id="767cbab4-a256-1f74-2c1e-16d9066a3a47">
      <label text="cytosol"/>
      <bbox h="249.94285867622307" w="236.15252777792801" y="727.7768230361996" x="417.8363755059461"/>
    </glyph>
    <glyph class="macromolecule" id="ba413b6e-b420-164c-471b-146b4e9375ec" compartmentRef="767cbab4-a256-1f74-2c1e-16d9066a3a47">
      <label text="RAS"/>
      <bbox h="35" w="70" y="882.2856616515703" x="419.46137550594614"/>
      <glyph class="state variable" id="ba413b6e-b420-164c-471b-146b4e9375ec_0">
        <state value="inactive"/>
        <bbox h="28" w="60" y="903.2856616515703" x="448.96137550594614"/>
      </glyph>
    </glyph>
    <glyph class="complex" id="5791e1c0-a1ad-1767-c5e2-2c134ba3fb1f" compartmentRef="767cbab4-a256-1f74-2c1e-16d9066a3a47">
      <label text="SOS"/>
      <bbox h="82.70000000000005" w="73.25" y="739.4018230361996" x="470.14232743678065"/>
      <glyph class="macromolecule" id="f54d4c49-7e93-9350-6adb-9ed524682adc">
        <label text="SOS"/>
        <bbox h="35" w="70" y="741.0268230361996" x="471.76732743678065"/>
      </glyph>
      <glyph class="macromolecule" id="45f2f84d-3495-62ab-aff7-8a8f63f828e1">
        <label text="Grb2"/>
        <bbox h="35" w="70" y="785.4768230361997" x="471.76732743678065"/>
        <glyph class="state variable" id="45f2f84d-3495-62ab-aff7-8a8f63f828e1_0">
          <state value="active"/>
          <bbox h="12" w="30" y="779.4768230361997" x="505.76732743678065"/>
        </glyph>
      </glyph>
    </glyph>
    <glyph class="simple chemical" id="e49a9d1b-2180-3079-fa85-485090605535" compartmentRef="767cbab4-a256-1f74-2c1e-16d9066a3a47">
      <label text="GTP"/>
      <bbox h="35" w="35" y="941.0946817124227" x="495.33778652650176"/>
    </glyph>
    <glyph class="process" id="53b5ff75-22e0-c178-12cf-d42a65d2ab13" compartmentRef="767cbab4-a256-1f74-2c1e-16d9066a3a47">
      <bbox h="10.714285714285715" w="10.714285714285715" y="891.8968864547796" x="530.6715767996084"/>
      <port id="53b5ff75-22e0-c178-12cf-d42a65d2ab13.1" y="897.2540293119225" x="528.5287196567513"/>
      <port id="53b5ff75-22e0-c178-12cf-d42a65d2ab13.2" y="897.2540293119225" x="543.5287196567513"/>
    </glyph>
    <glyph class="simple chemical" id="dbe0e869-9afe-1608-a48f-f0d40f533732" compartmentRef="767cbab4-a256-1f74-2c1e-16d9066a3a47">
      <label text="GDP"/>
      <bbox h="35" w="35" y="933.6623216106963" x="565.4884400543842"/>
    </glyph>
    <glyph class="macromolecule" id="2c70ba44-45dd-abab-f7cd-125c489ca0ae" compartmentRef="767cbab4-a256-1f74-2c1e-16d9066a3a47">
      <label text="RAS"/>
      <bbox h="35" w="70" y="865.1201142403494" x="582.3639032838742"/>
      <glyph class="state variable" id="2c70ba44-45dd-abab-f7cd-125c489ca0ae_0">
        <state value="active"/>
        <bbox h="12" w="30" y="859.1201142403494" x="616.3639032838742"/>
      </glyph>
    </glyph>
    <arc class="consumption" id="70c9d3ec-7d39-9c30-b2e8-b4a2e10a12ca" target="53b5ff75-22e0-c178-12cf-d42a65d2ab13.1" source="e49a9d1b-2180-3079-fa85-485090605535">
      <start y="941.2901084162036" x="519.3800906340604"/>
      <end y="897.2540293119225" x="527.9037196567513"/>
    </arc>
    <arc class="production" id="7060d150-286b-b10d-3833-152a346c567b" target="dbe0e869-9afe-1608-a48f-f0d40f533732" source="53b5ff75-22e0-c178-12cf-d42a65d2ab13.2">
      <start y="897.2540293119225" x="544.1537196567513"/>
      <end y="934.8564062403929" x="568.7842945701777"/>
    </arc>
    <arc class="stimulation" id="e1b2b837-dba4-cb0e-edd5-38e19f5df684" target="53b5ff75-22e0-c178-12cf-d42a65d2ab13" source="5791e1c0-a1ad-1767-c5e2-2c134ba3fb1f">
      <start y="832.6018230361997" x="519.7902841266716"/>
      <end y="888.4195959151783" x="533.8098104281008"/>
    </arc>
    <arc class="consumption" id="1a64543c-9019-8d6e-a3ea-b1e5edee7177" target="53b5ff75-22e0-c178-12cf-d42a65d2ab13.1" source="ba413b6e-b420-164c-471b-146b4e9375ec">
      <start y="898.6838365401466" x="489.96137550594614"/>
      <end y="897.2540293119225" x="527.9037196567513"/>
    </arc>
    <arc class="production" id="941945f9-390d-2edf-3c23-c78f08f276b0" target="2c70ba44-45dd-abab-f7cd-125c489ca0ae" source="53b5ff75-22e0-c178-12cf-d42a65d2ab13.2">
      <start y="897.2540293119225" x="544.1537196567513"/>
      <end y="889.5606810574964" x="578.7882880921784"/>
    </arc>
  </map>
</sbgn>
```


Figure 2.3: SBGN-ML Sample

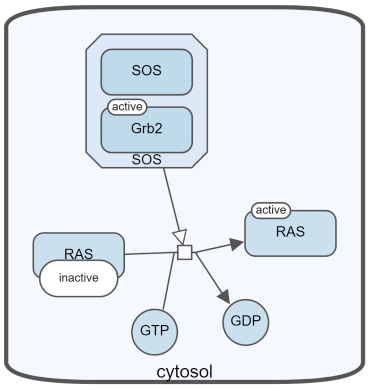


Figure 2.4: Corresponding SBGN diagram of the SBGN-ML sample in Figure 2.3

2.3 Neo4j

Neo4j is a complete graph database management system that was implemented in Java. It uses property graphs as data model which contains nodes, relationships and properties. Therefore, it is very suitable for highly connected, heterogeneous and complex data [12]. It is suitable for highly connected data because it does not require join operations to relate data. Since it stores and prioritizes relationships as much as it stores and prioritizes nodes, Neo4j uses stored relationships to perform traversals on the graph instead of using costly join operations to connect data for traversal.

Since Neo4j is a complete database management system, it allows users to run all CRUD (create, read, update and delete) operations on the graph data in the database. In order to perform such operations, Cypher, which is Neo4j's proprietary query language, can be used (Figure 2.7). Like other graph database systems, you can add stored procedures to Neo4j.

Neo4j browser has a user interface for managing, querying, styling, monitoring, and visualization of database contents (Figure 2.5). It provides an editor for the database users to run Cypher statements and to visualize results as a rendered graph or as a table. Style of the rendered graph can be customized in terms of size and color.

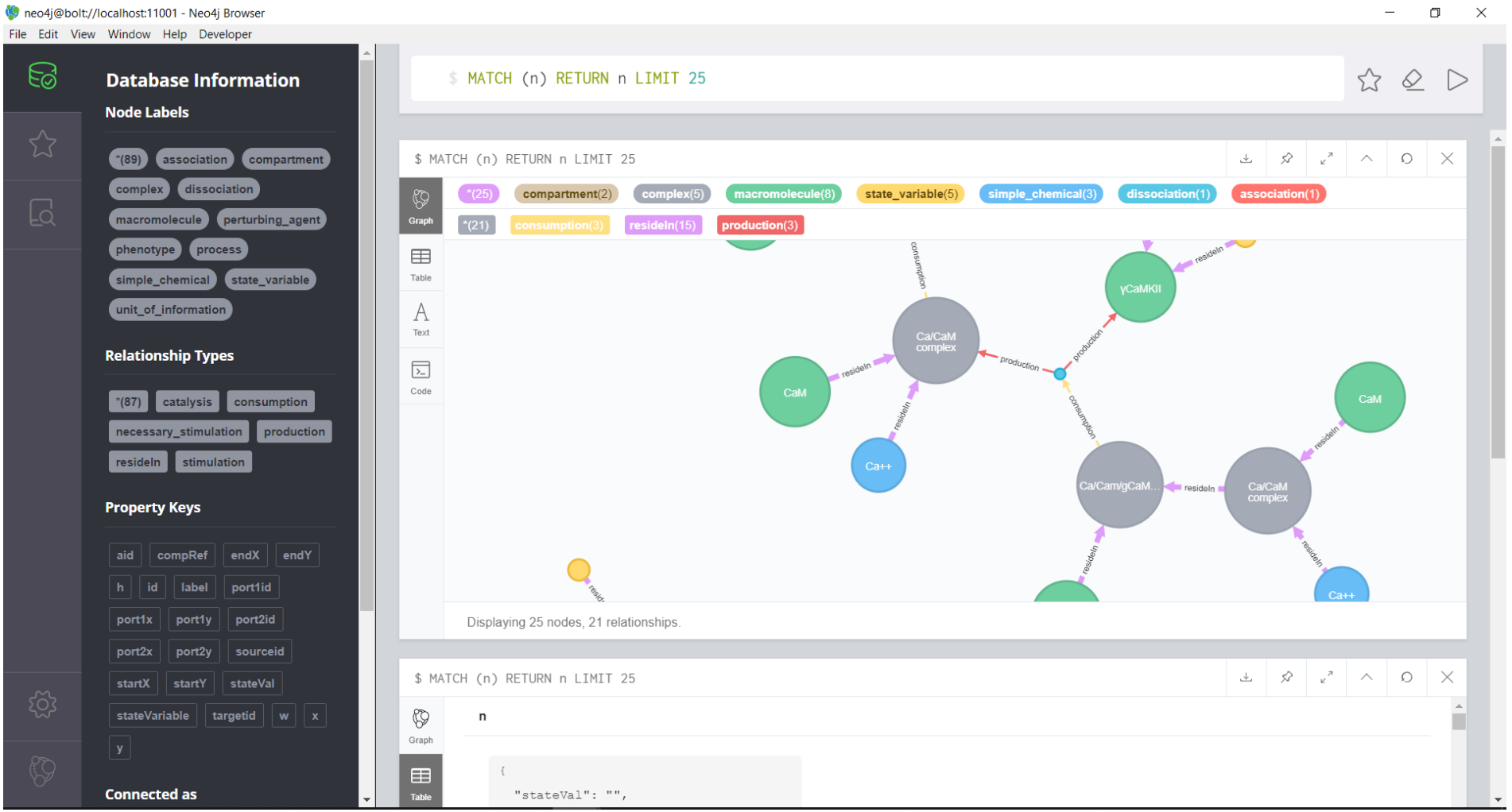


Figure 2.5: Neo4j Browser

2.3.1 Property Graph Data Model

Neo4j uses the property graph as its data model. The property graph is a graph whose nodes and relationship may have properties. You can see an example property graph data model in Figure 2.6. Property graph data model has three basic elements which are node, relationship and property [13, 14].

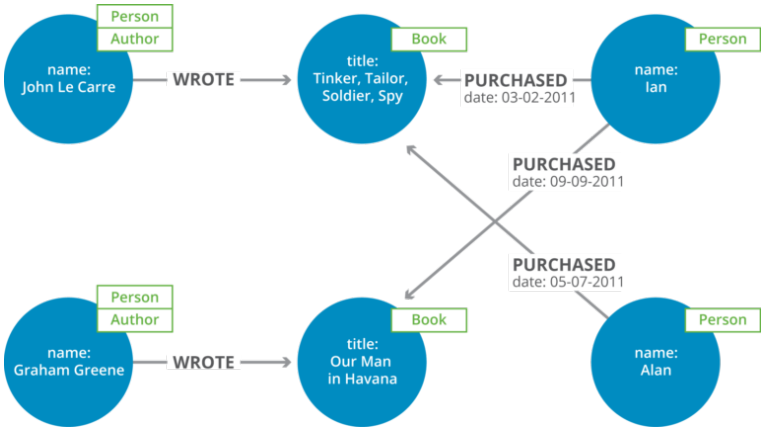


Figure 2.6: Property graph data model example for book store [2]

Entities of the graph is represented by *nodes*. In order to name nodes, *labels* are used. Nodes can be disconnected or connected to other nodes by *edges*. *Attributes* can be attached to nodes.

A *relationship* or an edge is between source and target nodes. Relationship in the property graph must have *direction*, *source*, *target* and *name*. There is no support for undirected relationships however need for undirected edges can be handled by ignoring direction. Similarly, attributes can be attached to relationships and *labels* are used to name relationships.

Properties are key value-pairs to attach data to both nodes and relationships [13, 14, 15]. Properties give countenance to various types such as number, string, boolean, spatial and temporal.

2.3.2 Cypher

Cypher language which is Neo4j's own query language for database operations is composed of ASCII characters (Figure 2.7) [16]. It is a declarative query language that lets users efficiently query and manipulate data, which is stored as graph structure in a graph database [17]. It is advantageous for users who work on domains consisting highly connected data, because traversals can be done via relationships instead of having to perform join operations.

```
match (p:process)
where p.id = {id}
optional match p2=(p)-[]->(b)
optional match p4=(b)-[:resideIn*]-(d)
return [b.id,d.id] as idList,  [p2,p4] as pathList
```

Figure 2.7: Example Cypher statement for retrieving 1-neighbourhood of a process node.

Furthermore, it is a pattern matching language, where patterns can be defined with ASCII characters [17]. Being a pattern matching language makes Cypher let users to traverse path, relationship or node by defining expected pattern. Although all relationship in property graph data model must have direction, it is possible to write Cypher statements without indicating whether direction is ingoing or outgoing. By Cypher statements, you can return path, node, relationship or property. Structure of Cypher language is similar to the structure of SQL [17]. Most of Cypher clauses are similar to SQL clauses.

2.3.3 User-Defined Procedure

A user defined procedure in Neo4j is similar to stored procedures of other database systems. The user can design their own function for a specific purpose as a procedure. After users add the procedure to their database, they can call the procedure to achieve their specific purpose with given parameters to return results. User

defined procedures of Neo4j are implemented in Java. After the procedure is implemented, it is packaged as a .jar file, and manually needs to be moved to the plugin folder of local Neo4j installation. This way the procedure becomes available for database users for execution as a Cypher statement.

When implementing user defined procedures in Java, in order to make database operations or execute Cypher statements, Neo4j's official Java driver is used. Java driver of Neo4j is available via Maven [18]. After deployment of user defined procedures, any application may call these procedure via the specified protocol to execute them on Neo4j's execution engine (Figure 2.8).

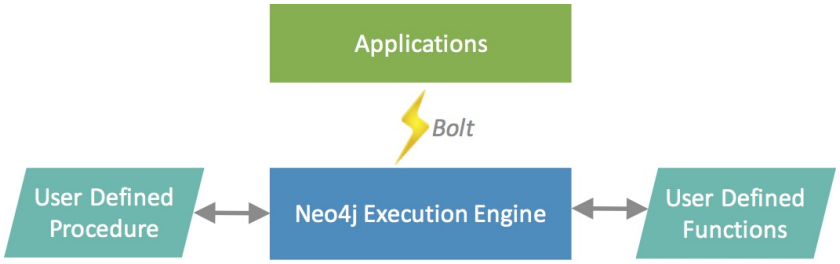


Figure 2.8: Workflow diagram for user defined procedures [3]

Furthermore, user defined procedures of Neo4j provide flexibility for developers to add third party dependencies or libraries to Neo4j not supported by Cypher [8]. For example, libSBGN can be added as a procedure in order to manipulate SBGN maps [7].

2.4 Cytoscape

Cytoscape.js is an open source software library for visualization of networks. Even tough it is widely used in the biological domain (Figure 2.9), it is not only applicable to biological networks but also applicable to various domains including social networks. Cytoscape.js supports directed/undirected simple/compound graphs. It provides functions for various graph traversals and pre-defined operations as well as basic import/export and automated layout. Furthermore, Cytoscape.js

can be extended by an extension (plugin) mechanism to provide new features [4], such as cytoscape-cose-bilkent for automated layout [19] and cytoscape-expand-collapse for collapsing and expanding compound structures to manage complexity of large graphs [20].

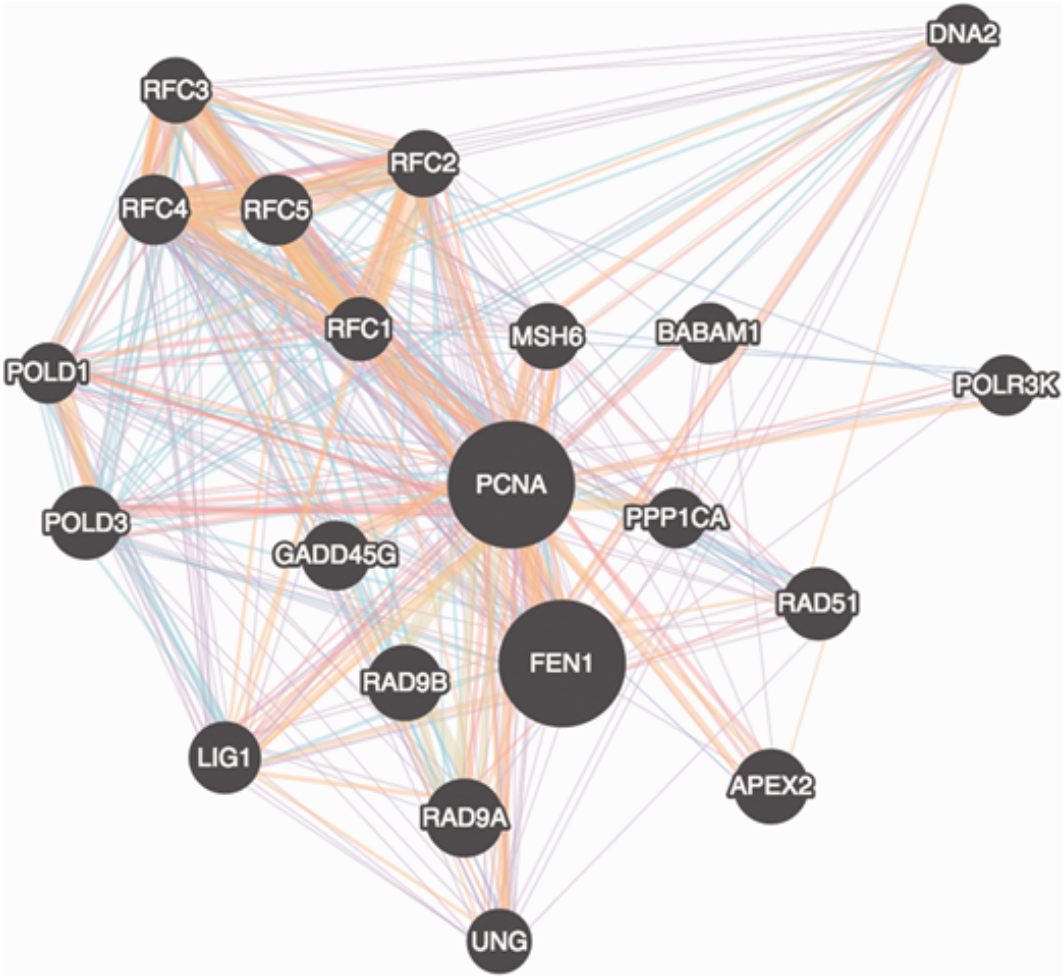


Figure 2.9: A biological network visualization by Cytoscape [4]

2.5 Newt

Newt is a web based open source visualization tool for editing and viewing SBGN diagrams (Figure 2.11). It is implemented by using Cytoscape.js based libraries SBGNViz.js and ChiSE.js (Figure 2.10). SBGNViz.js and ChiSE.js are libraries to

respectively visualize and edit SBGN diagrams in process description and activity flow languages.

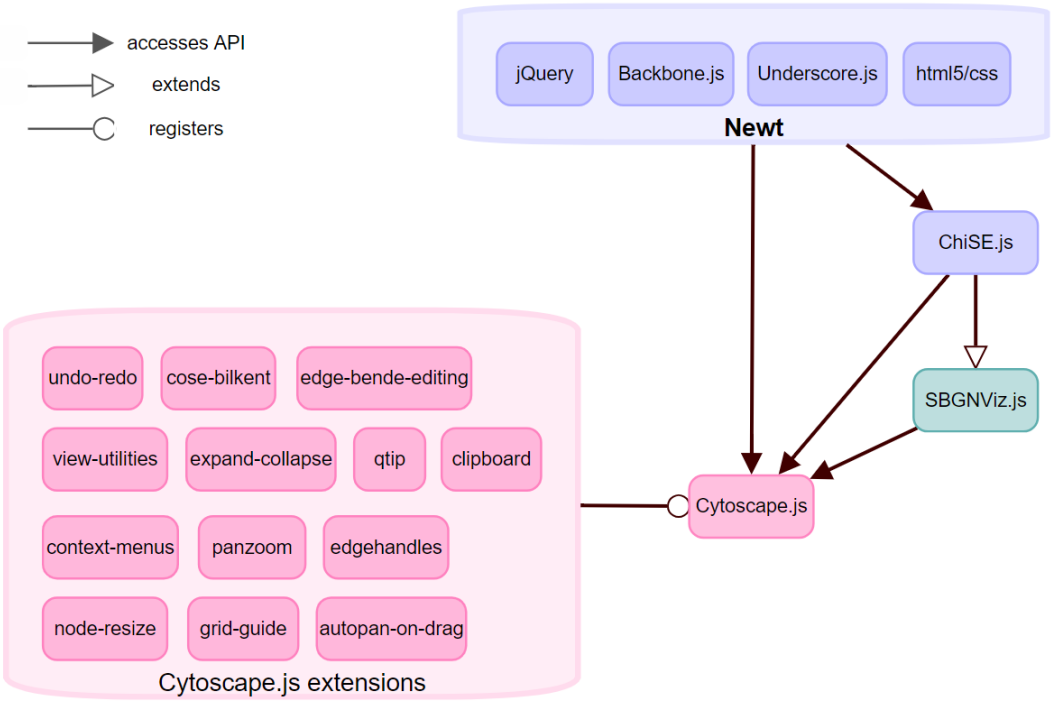


Figure 2.10: Architecture of Newt [5]

By derived features from Cytoscape.js, SBGNViz.js, ChiSE.js and Newt, users can [5]

- Import and export SBGN maps from and to an SBGN-ML file.
- Create and edit an SBGN map.
- Highlight nodes and relationship.
- Expand and collapse compound nodes.
- Style an SBGN map
- Layout an SBGN map.

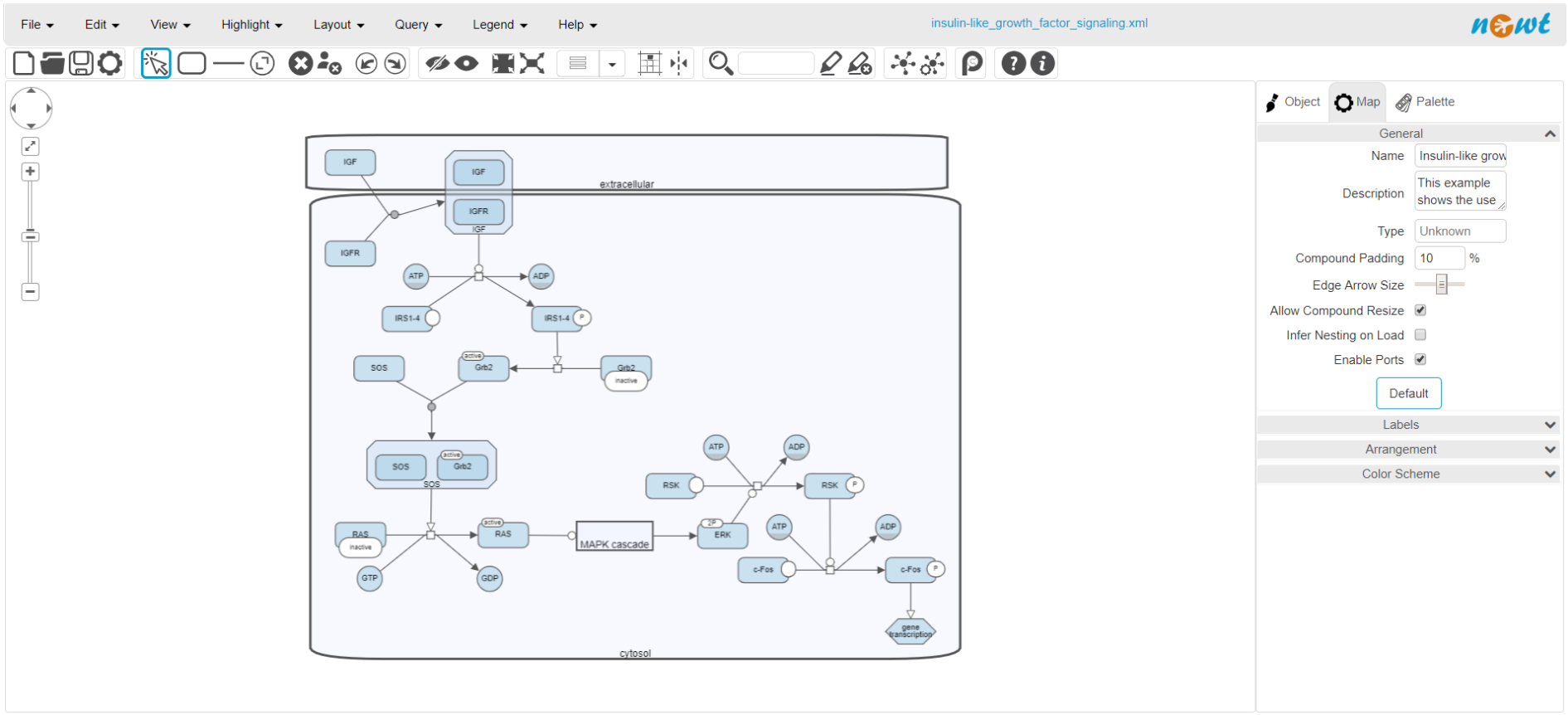


Figure 2.11: Sample screenshot from the Newt Editor

2.6 STON

STON is a Java based tool for exporting SBGN maps into Neo4j graph database [10]. Furthermore, it merges two SBGN maps by comparing relationships of process nodes and nodes at the other end of these relationships [10]. In order to store SBGN maps in the Neo4j graph database, STON respectively maps glyphs and arcs of SBGN to nodes and relationships of the graph database.

Chapter 3

Graph Based User Defined Procedures for SBGN Maps

3.1 Data Model

In order to model SBGN maps in Neo4j graph database, we take the format of SBGN-ML file as skeleton. The simple reason for this is its similarity to property graph data model which enables us to directly map elements of SBGN maps (glyphs, arcs and attributes) to elements of property graph (nodes, relationships, and properties, respectively).

A glyph in an SBGN-ML file refers to a node of property graph data model. In order to model a glyph in the property graph (See Table 3.1):

- *class* attribute of the glyph is mapped to *label* of the node.
- *id* attribute of the glyph is mapped as *id* property of the node.
- *compartmentRef* attribute of the glyph is mapped as *compRef* property of the node
- *text* attribute of label element of the gylph is mapped as *label* property of

the node.

- *variable* and *value* attributes of state element of the glyph are respectively mapped as *stateVariable* and *stateVal* properties of the node.
- *x*, *y*, *w* and *h* attributes of bbox element of the glyph are respectively mapped as *x*, *y*, *w* and *h* properties of the node.
- *id*, *x* and *y* attributes of first port element of the glyph are respectively mapped as *port1id*, *port1x* and *port1y* properties of the node.
- *id*, *x* and *y* attributes of second port element of the glyph are respectively mapped as *port2id*, *port2x* and *port2y* properties of the node.

Glyph	Node
class	Label
compartmentRef	property [compRef]
label	property [label]
state value	property [stateVal]
state variable	property [stateVariable]
id of first port	property [port1id]
id of second port	property [port2id]
x of first port	property [port1x]
x of second port	property [port2x]
y of first port	property [port1y]
y of second port	property [port2y]
h of bbox	property [h]
w of bbox	property [w]
y of bbox	property [y]
x of bbox	property [x]

Table 3.1: Mapping from a glyph to a node

An arc in an SBGN-ML file corresponds to a relationship in property graph data model. While modeling the arc in the property graph (See Table 3.2):

- *class* attribute of the arc is mapped to *label* of the relationship.
- *id* attribute of the arc is mapped as *id* property of the relationship.

- x and y attributes of start element of the arc are respectively mapped as *startX* and *startY* properties of the relationship.
- x and y attributes of end element of the arc are respectively mapped as *endX* and *endY* properties of the relationship.
- *source* and *target* attributes of the arc are respectively mapped as *sourceid* and *targetid* properties of the relationship.

Arc	Relationship
class	Label
id	property [id]
target	property [targetid]
source	property [sourceid]
x of start	property [startX]
y of start	property [startY]
x of end	property [endX]
y of end	property [endY]

Table 3.2: Mapping from an arc to a relationship

While respectively mapping an arc and a glyph to a relationship and a node, all values of properties are directly mapped or a few format issues applied. In order to relate complexes to their member child nodes, we define *resideIn* relationship between parent node and child node.

Our mapping rules to create a SBGN data model are similar to mapping rules of STON. The difference of our data model is that we model compartment, state-variable and unit-of-information as separate nodes while STON models them as node properties. We add a node for each state-variable and unit-of-information of a glyph. However, STON concatenates information about all state-variable elements of a glyph into a single property and information about all unit-of-information elements of a glyph into another single property to attach these properties to the node. Adding a node for each state-variable/unit-of-information instead of modeling them as a property results in a more crowded database. However, if we insert a node for each state-variable/unit-of-information, we can

use state-variable or unit-of-information in Cypher statements while defining a graph pattern by ASCII characters as exemplified in Figure 3.1.

```
Match (a:macromolecule)-[]-(b:state_variable {stateVariable: 'T287', stateVal: 'P'}),
(a)-[]-(c:state_variable{stateVariable:'S334', stateVal: 'P'})
return a
```

Figure 3.1: Cypher statement to get macromolecules with state-variable P@T287 and state-variable P@S334

3.2 Procedures on Cypher for Query Algorithms

Neo4j allows database users to write user defined procedures to take advantage of third party Java libraries and combine with Neo4j's own Java driver. This way after deployment of user defined procedures, features of third party library become available to call by Cypher query. In this thesis, in order to combine features of Neo4j with libSBGN, we implement user defined procedures that can be called via Cypher queries.

Specifically we implement user defined procedures for both graph query algorithms, which are defined by Dogrusoz et al. [21], and helper functions. Implemented graph algorithm procedures are *Neighbourhood*, *Common Stream*, *Paths-From-To*, and *Paths-Between*. Helper procedures are *Insert Graph* and *Read From Graph DB*.

3.2.1 General Rules and Methods

Our graph based user defined procedures share some common features and rules.

- Procedures take graph/sub-graph as SBGN-ML text if needed.

- Procedures return graph/sub-graph as SBGN-ML text if needed.
- If at least one process node is in the result, its input and output should also be included in the resulting sub-graph. For example, if the result set Figure 3.2 contains only highlighted nodes (A, C, F and process nodes) and relationships among them, nodes (B, D, E, F) and their relationships to process nodes should also be returned since without these nodes resulting SBGN map would be incomplete/invalid.
- If a member of a compound (complex or compartment) is in the result, returned sub-graph should also contain its parent compound node.
- If a complex is in the result, returned sub-graph should also include members of the complex. However, if a compartment is in the result, we should not include any of its members.
- We take graph theoretical distances or simply lengths differently for SBGN graphs that are essentially bipartite graphs (process nodes and EPNs) for convenience. For instance, if there is a path, where an edge goes to a process node and then another edge from the process node to another EPN, we take the length of this path as 1, although the graph theoretical length of the path is 2. In other words, we only count process nodes in length calculations. For example in Figure 3.2, despite the fact that the number of relationships between A and C is 2, length of this path is 1 according to our definition of length as A goes to process node before it goes to C. Furthermore, length of path which is between A and F is 2 due to our definition of length.

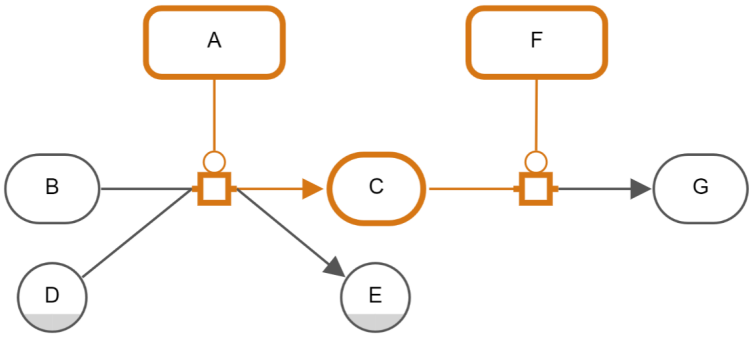


Figure 3.2: Sample result path (highlighted) and the extended network that is intact as taken from Newt

Implemented user defined procedures with Cypher language for graph algorithms sharing a common traversal method with minor differences. We designed our *compound based breadth first traversal method* according to SBGN data model by using Cypher statements with defined patterns. Compound based breadth first traversal method takes aforementioned rules for compound nodes and revised definition of length/distance into consideration.

```
match (a)
where a.id = {id}
optional match (a)-[:resideIn*]-(c)
return  collect(a.id) as idlist, collect(c.id) as cidlist
```

Figure 3.3: Cypher statement for getting a compound node and its members including parent, child and siblings.

In order to handle compound nodes, Cypher statements with patterns are used. While defining these patterns for retrieving members of a compound, compound (parent) for a member or siblings, *resideIn* relationship which relates children to their parents in data model is used. Specifically, the Cypher statement in Figure 3.3 is for this purpose. It starts with *Match (a)* and continue with *where a.id = id*. This means that match a node(a) whose id is *id*. Then, it uses *optional*

match clause to indicate that match if specified pattern exist. *(a)-[:resideIn*]-(c)* means that match all existing paths starting from the node(a) to any node(c) and these relationships of the paths are restricted to be labelled with *resideIn*. * symbol after *resideIn* indicates that there is no length limit for paths whose relationships are labelled with *resideIn*. In other words, *(a)-[:resideIn*]-(c)* recursively retrieves all parent, child, sibling of node(a) as node(c). Then, *collect(c.id) as cidlist* statement collects ids of node(c) into a list for next traversal. As compound traversal proceeds, we check whether or not a visited node is a process. If the reached node is a process node, 1-neighbourhood of the process node is added to reached node set by a Cypher statement (Figure 2.7).

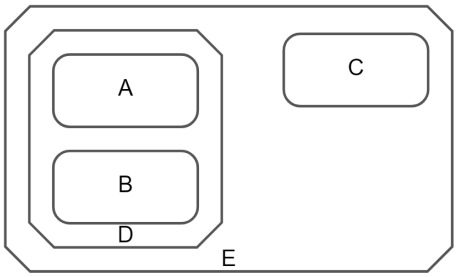


Figure 3.4: Sample compound graph containing nested complexes and macromolecules

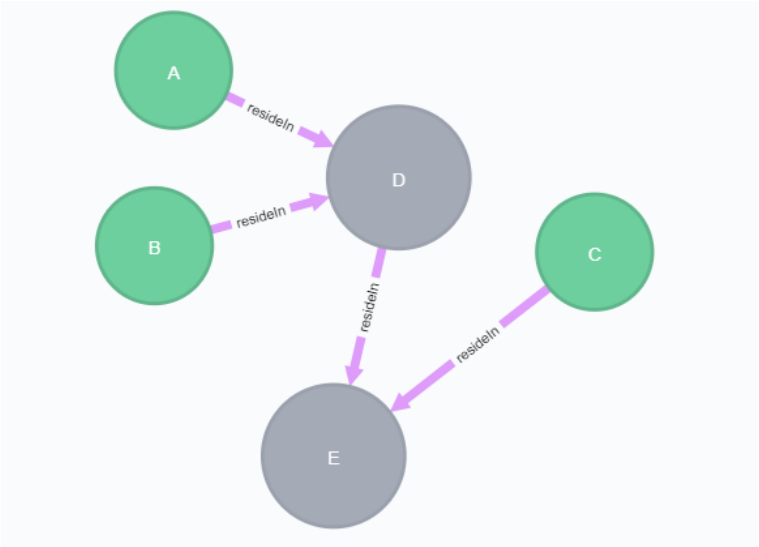


Figure 3.5: Visualization of the compound graph in Figure 3.4 as represented by Neo4j data model.

The nested compound graph in Figure 3.4 contains complex nodes (D, E) and macromolecules (A, B, C). If we insert SBGN map in Figure 3.4 into Neo4j according to SBGN data model, Neo4j browser renders this graph as in Figure 3.5. While relationship between parent and child node is represented by pink *resideIn* relationship in rendered graph, macromolecules and complexes are respectively represented by green and grey nodes. If we take id of any complex or macromolecule of the nested compound graph in order to give this id to Cypher statement in Figure 3.3, cidlist will contain ids of all nodes regardless of whether or not the given id belongs to outmost node or innermost node.

3.2.2 Populating Graph Database from an SBGN Map

For the purpose of inserting an SBGN map to Neo4j graph database, a user defined procedure named *Insert Graph* is implemented. This procedure takes the content of the SBGN-ML file as a string parameter. Then, it creates an SBGN object by using the libSBGN library. Subsequently, by using the SBGN object map created, all glyph and arc objects of the map are inserted into the Neo4j graph database in accordance with SBGN data model (Table 3.1 and Table 3.2).

For instance, should the SBGN map in Figure 3.6 be inserted to Neo4j graph database by *Insert Graph* procedure, Neo4j browser renders it as in Figure 3.7.

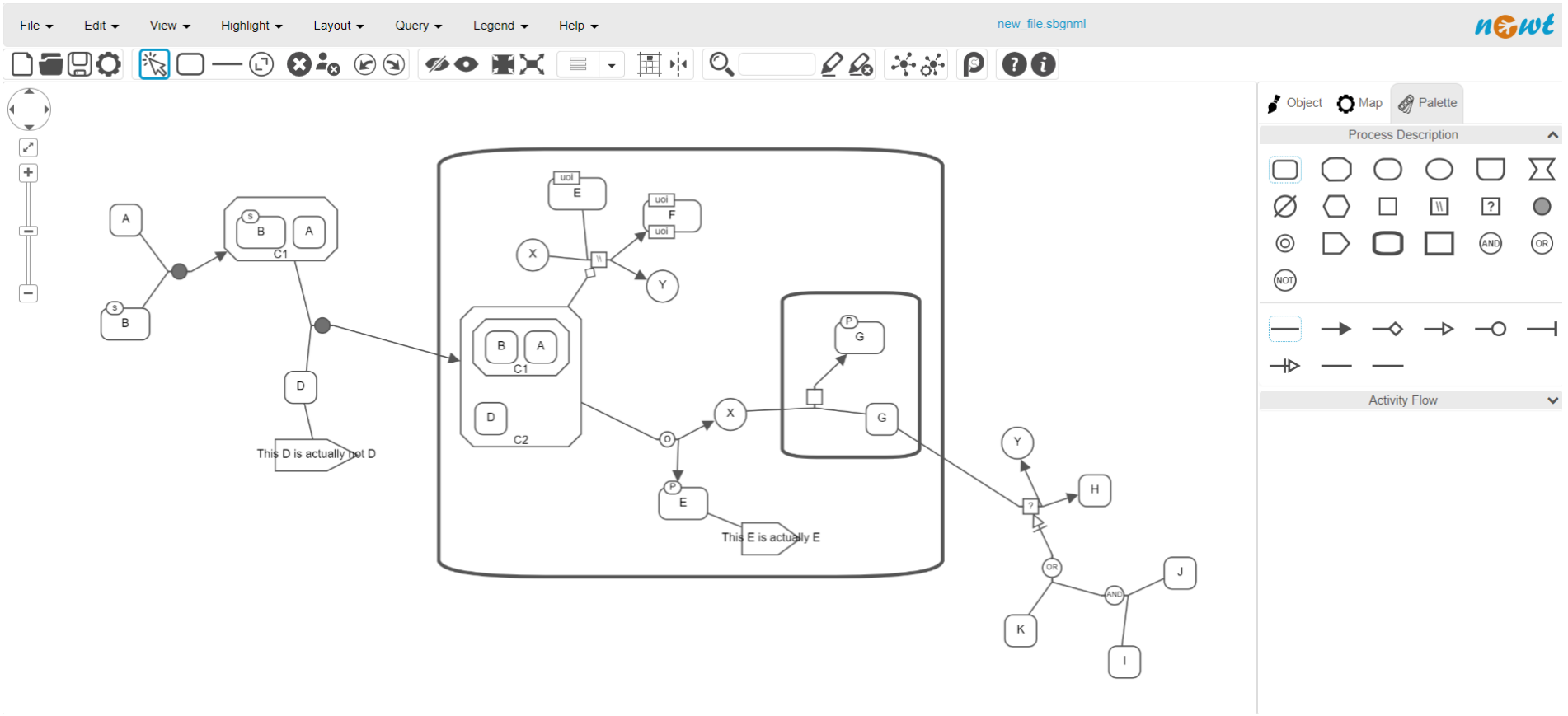


Figure 3.6: Visualization of a sample SBGN map in Newt

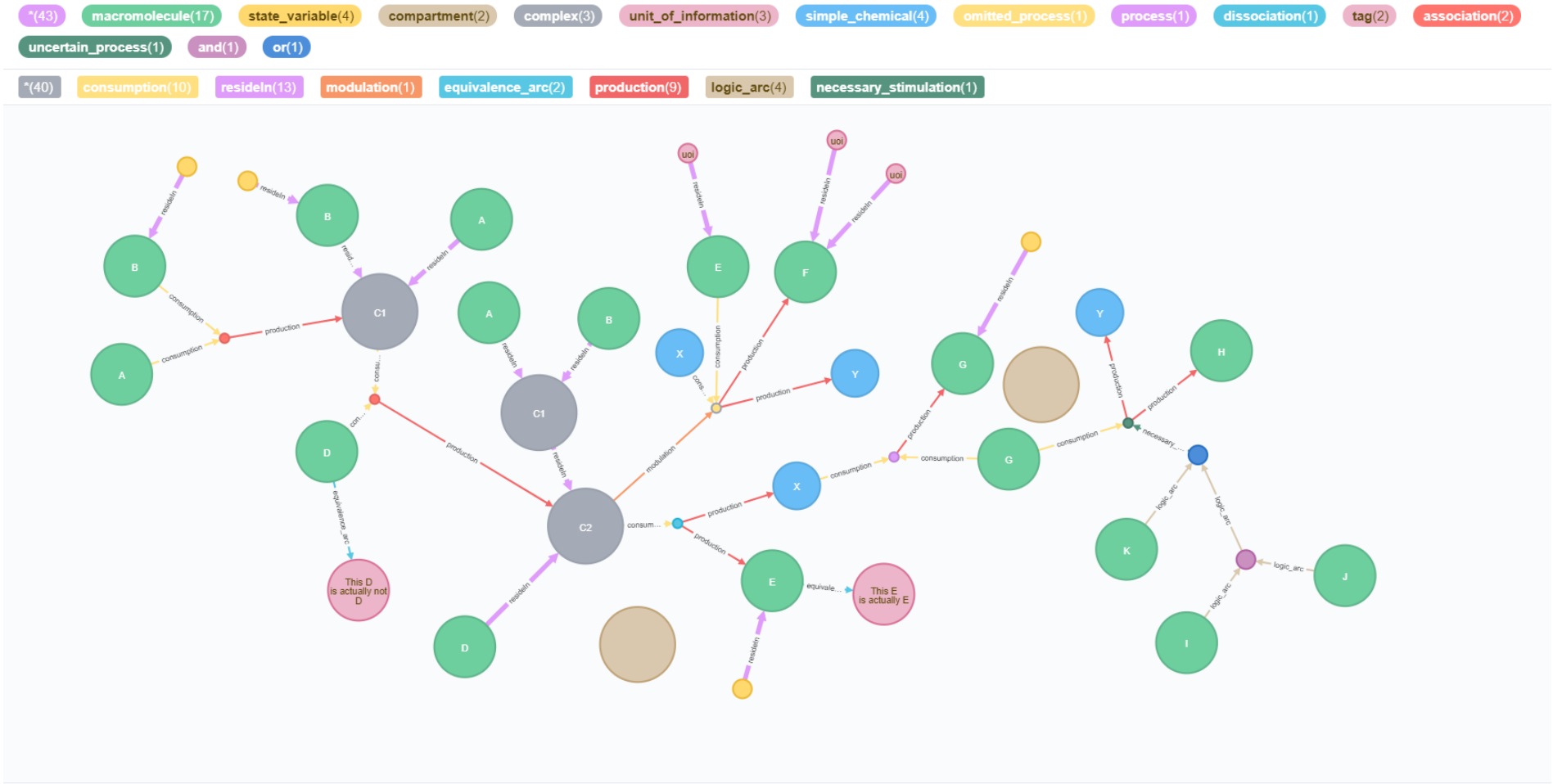


Figure 3.7: How sample SBGN map in Figure 3.6 is rendered by Neo4j browser in line with the data model

3.2.3 Loading an SBGN Map from the Database

Read From Graph DB is a user defined procedure to convert the graph stored in Neo4j graph database to SBGN-ML text as string. This procedure retrieves all nodes and relationships in the database by executing the Cypher statement in Figure 3.8 for retrieving nodes and the one in Figure 3.9 for retrieving relationships. Then, each retrieved node is mapped to a glyph by considering parent-child relationship between glyphs while each retrieved relationship is mapped to arc.

```
Match (g)
optional match (g)<-[:resideIn]-(ch)
return g, collect(ch) as childrenGlyph order by g.id
```

Figure 3.8: Cypher statement for retrieving all nodes from graph db

```
MATCH (a)-[r]->(b)
where r.startX is not null
return r.sourceid as sid, r.targetid as tid, r.startX as xx,
r.startY as yy, r.endX as ex, r.endY as ey, r.aid as id, type(r) as class
```

Figure 3.9: Cypher statement for retrieving all relationships from graph db

3.2.4 Neighbourhood Procedure

Neighbourhood query is the most basic algorithm that we implement from [21]. We develop a user defined procedure for this algorithm. It performs a breadth first traversal by using our traversal method that handles compound nodes by Cypher statement. Procedure takes two parameters which are a gene list and a length limit. It returns an object which contains SBGN-ML text.

The *Neighbourhood* procedure implements the algorithm for finding k-neighbourhood of given entities by making breadth first traversal. Starting from nodes whose labels are given as input, we traverse target nodes whose distance is 1 to a source node by using our traversal method. Then, target nodes become

source nodes. Until specified limit is reached, traversal continues. During traversal of target nodes, Cypher statements are used to traverse paths from the graph database in real time instead of caching all stored graph to run k-neighbourhood algorithm on a cached graph.

In order to return SBGN-ML text, visited paths are collected during traversal. Then, a nodes list and a relationships list are extracted from the collection of visited paths. Extracted list of nodes and list of relationships are respectively mapped to list of glyphs and list of arcs to create an SBGN map to return as SBGN-ML text

For instance, the *Neighbourhood* procedure is performed on the graph database that stores the SBGN map in Figure 3.10 to find 1-neighbourhood, 2-neighbourhood and 3-neighbourhood of macromolecule labeled U. Then, the resulting SBGN-ML map can be visualized by Newt as presented in Figure 3.11.

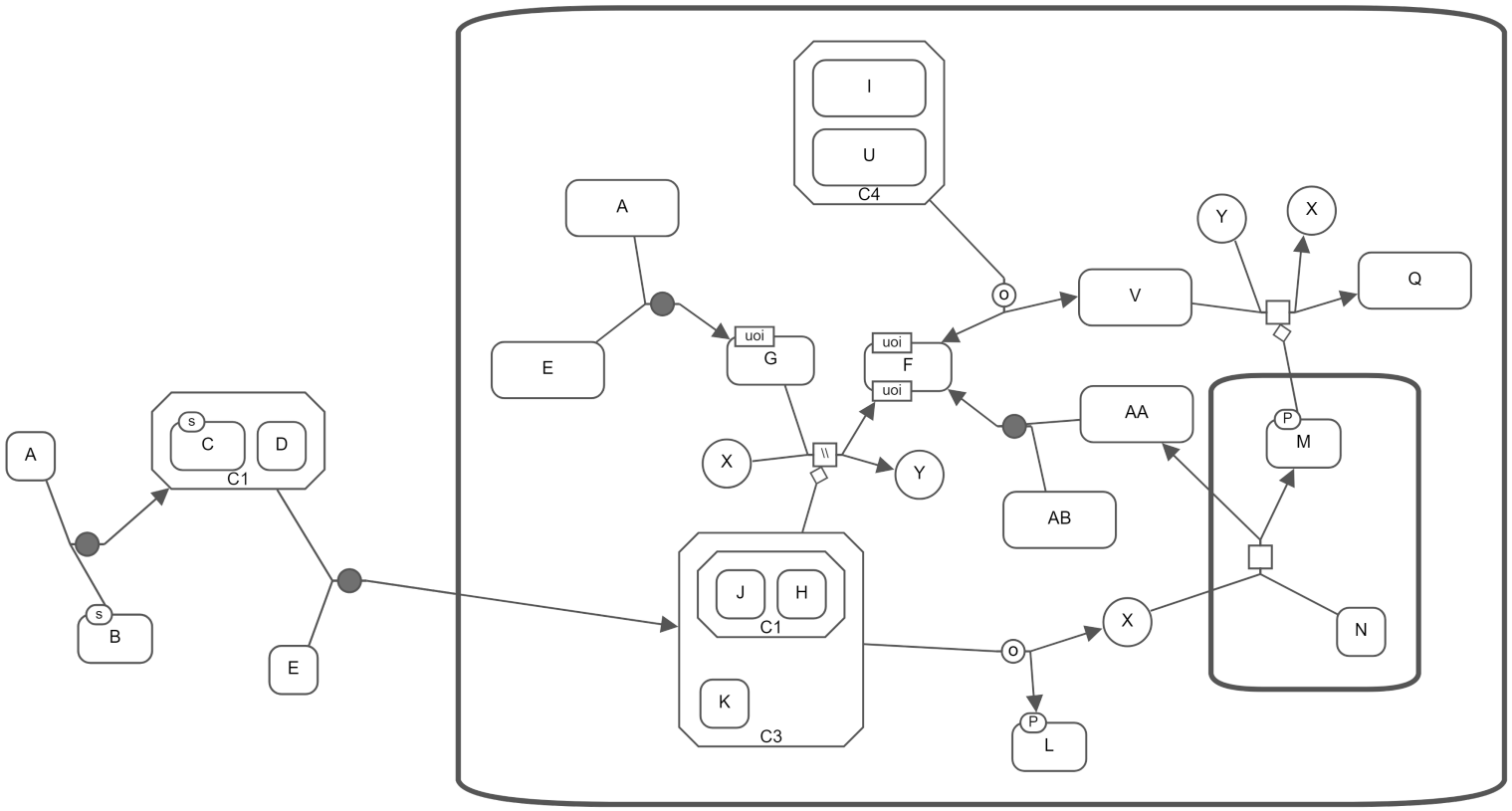
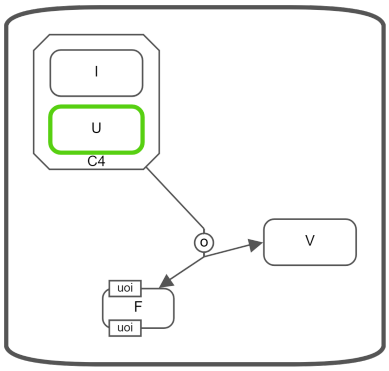
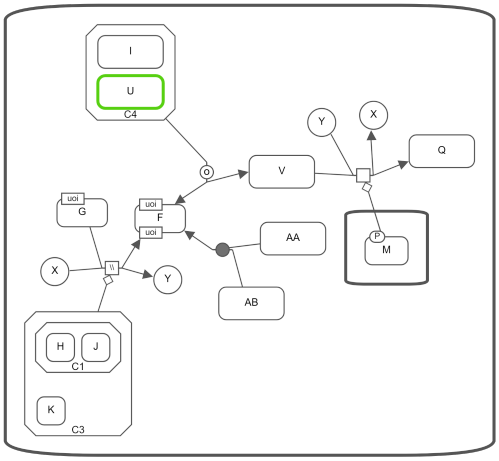


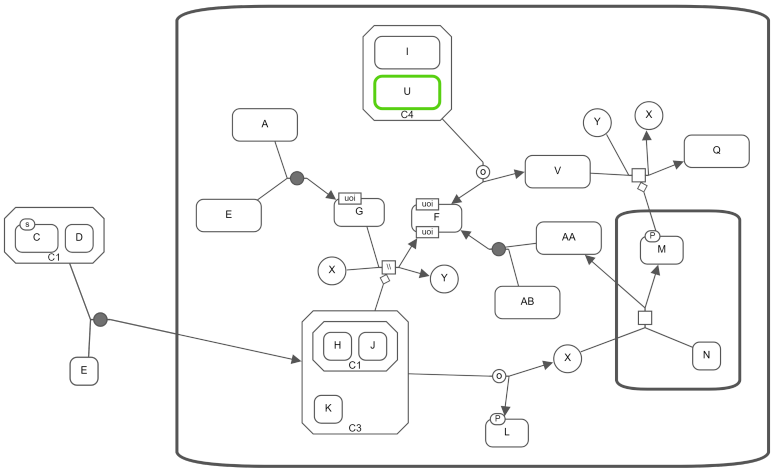
Figure 3.10: Sample graph in SBGN



(a)



(b)



(c)

Figure 3.11: Neighbourhood query on the SBGN map in Figure 3.10 (a) 1-neighbourhood of macromolecule U. (b) 2-neighbourhood of macromolecule U. (c) 3-neighbourhood of macromolecule U.

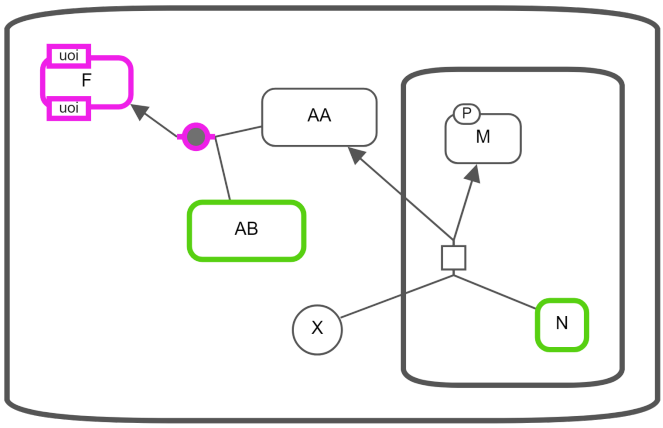
3.2.5 Common Stream Procedure

Common Stream is another user defined procedure that we have based on the *CommonUpstream/CommonDownstream algorithm* in [21]. Common up (down) stream algorithm finds all predecessors (successors) common to all input nodes. This procedure takes three parameters, such as gene list, limit and direction, and returns SBGN-ML text.

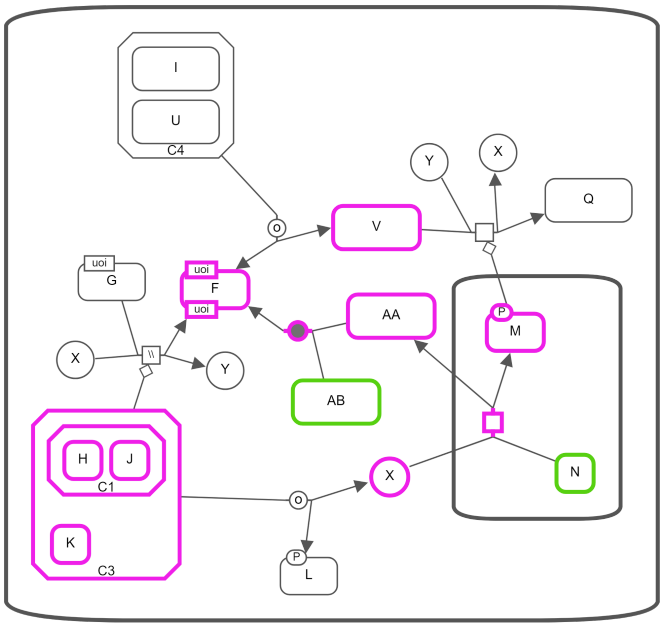
The direction parameter specifies direction of the traversal. If direction is 0, procedure traverses paths by using both incoming relationships and outgoing relationships. While direction 1 makes traversal of paths by using only outgoing relationships, direction 2 uses only incoming relationships for traversal of paths. Starting from each gene in the gene list, this procedure performs traversals to create a list of paths for each gene in the gene list by considering direction and length limit. Then, intersection of nodes of each input gene's path list gives us common stream nodes of input genes.

To create an SBGN map from the Neo4j sub-graph which contains paths from each common stream node to each input node in gene list, common stream nodes are used together with nodes and relationships of paths which are retrieved during traversal. Finally, the constructed SBGN map of the sub-graph is converted to SBGN-ML text.

Figures 3.12 and 3.13 show and example usage of this query. While input genes are highlighted in green, common nodes are highlighted in pink.

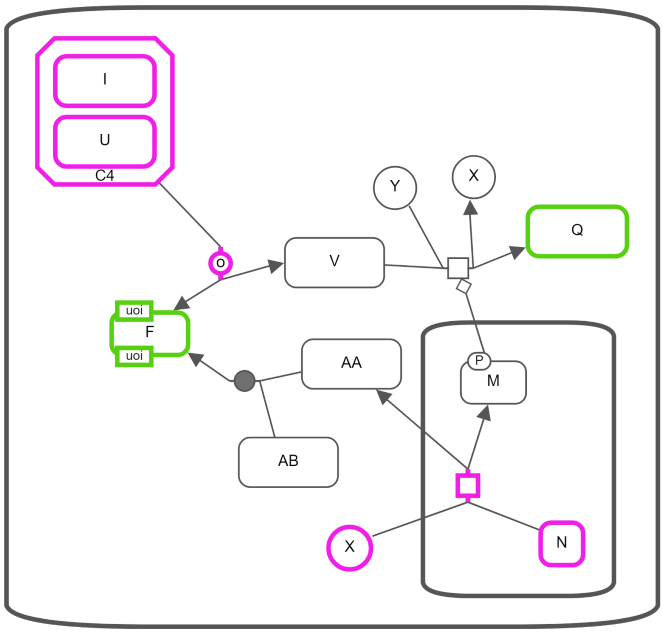


(a)

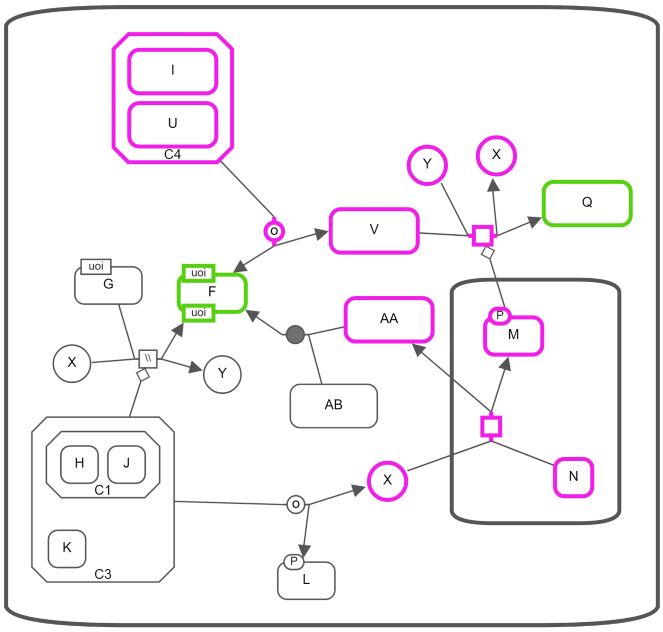


(b)

Figure 3.12: Common Stream query on the SBGN map in Figure 3.10 (a) Common nodes of AB and N with using outgoing relationships and length limit up to 2. (b) Common nodes of AB and N with using both incoming and outgoing relationships and length limit up to 2.



(a)



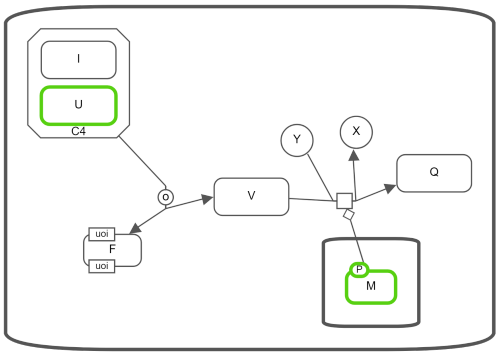
(b)

Figure 3.13: Common Stream query on the SBGN map in Figure 3.10 (a) Common nodes of Q and F with using incoming relationships and length limit up to 2. (b) Common nodes of Q and F with using both incoming and outgoing relationships and length limit up to 2.

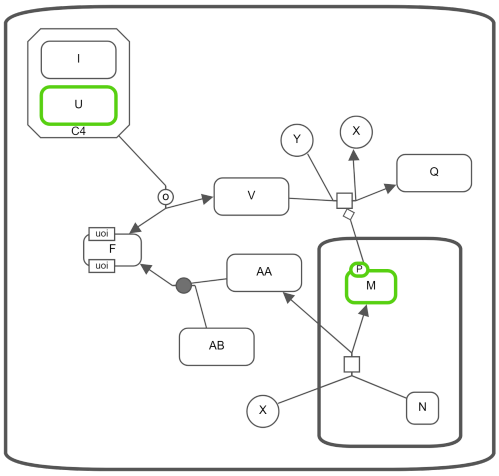
3.2.6 Paths-Between Procedure

Paths-Between is a user defined procedure that inspired from the *Graph of Interest (GoI) algorithm* mentioned in [21], where the goal is to compute a *minimal* sub-graph containing the input nodes and the paths connecting these nodes. In order for the traversal to work properly for this procedure, we edit our compound based breadth first traversal method according to the GoI algorithm. *Paths-Between* procedure takes two parameters which are genes list and limit.

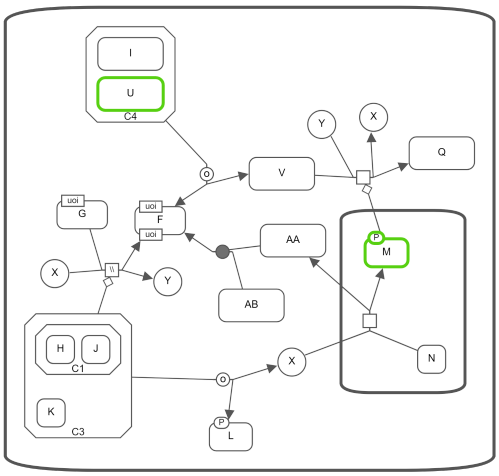
Procedure aims to find paths between entities in the input gene list up to the provided length limit. Starting from each gene, the graph is traversed up to the length limit. During traversal, nodes are weighted with their distance to the starting root gene and all reached paths are collected for purifying (creating sub-graph by using collected nodes and relationships) result graph. Then, the nodes whose total weight for any pair of genes less than or equal to the length limit are added to the result set. In other words, if sum of a node's distance to any two input genes is not greater than the limit, the node should be added to the result list. After resulting list of nodes is finalized, nodes in genelist and nodes in the result set are combined into a single node list and returned as the final result sub-graph after getting combined with the relationships that are extracted from reached and collected paths (Figure 3.14 and Figure 3.15). Then, the purified sub-graph is mapped to an SBGN map to return SBGN-ML text with the help of the libSBGN library and using conversion rules which are presented in Table 3.1 and Table 3.2.



(a)

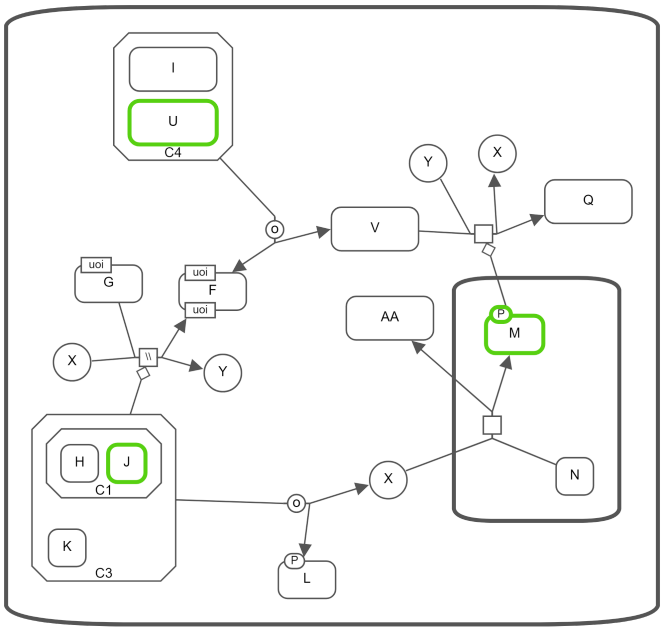


(b)

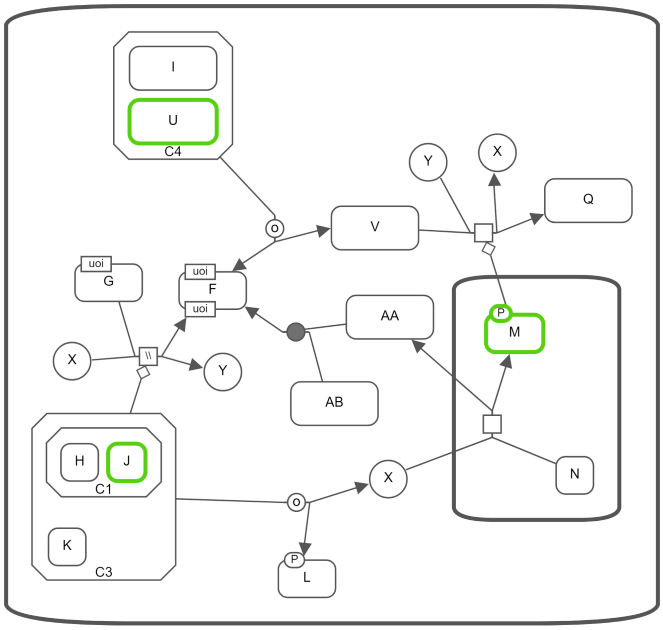


(c)

Figure 3.14: Paths-Between query on the SBGN map in Figure 3.10. (a) Paths between U and M with length limit up to 2. (b) Paths between U and M with length limit up to 3. (c) Paths between U and M with length limit up to 4.



(a)



(b)

Figure 3.15: Paths-Between query on the SBGN map in Figure 3.10. (a) Paths between U, M and J with length limit up to 2. (b) Paths between U, M and J with length limit up to 3.

3.2.7 Paths-From-To Procedure

Paths-From-To is another graph based user defined procedure that we implemented. It utilizes the *ShortestPaths algorithm* described in [21]. *Paths-From-To* has four parameters which are source list, target list, limit and additional distance. Similarly, it returns the result as SBGN-ML text.

Aim of *Paths-From-To* is finding all shortest paths between source genes and target genes with limited length of shortest path plus some user specified additional distance [21]. *limit* parameter defines when procedure should stop breadth first traversal. Also, additional *distance* parameter indicates extra length that procedure continues the traversal after shortest path between source list and target list has been reached.

Similar to *Neighbourhood*, *Common Stream* and *Paths-Between* procedures, graph traversal is done by compound based breadth first traversal method which handles complex structures by Cypher patterns with *resideIn* relationship and respects our definition of length. In order to purify paths after the traversal completes, unlike other procedures, for each visited node, it creates a structure containing nodes of the path between the root node and the visited node. By using these structures together with relationships which are also collected during the traversal, the resulting sub-graph that contains paths from source genes to target genes is created (Figure 3.16 and Figure 3.17). As with other procedures, the sub-graph is converted to an SBGN map for returning as SBGN-ML text.

In Figure 3.10, the length of a shortest path from G and K to L and Q is 1. Therefore, *Paths-From-To* procedure returns paths whose length is 1 plus value of additional distance parameter. If we set the additional distance parameter as 0, the procedure only returns those paths whose length is 1 (Figure 3.16a). If the value of this parameter is 1, on the other hand, paths whose length is up to 2 are returned (Figure 3.16b).

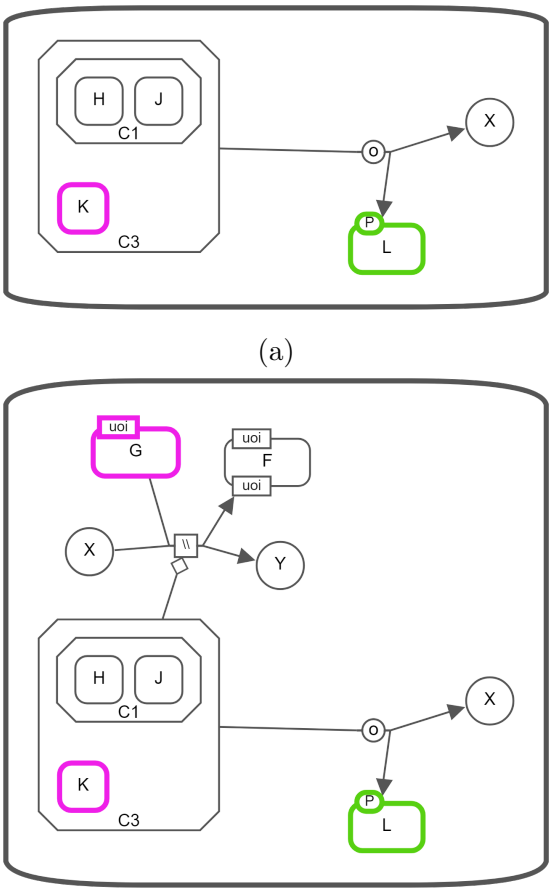
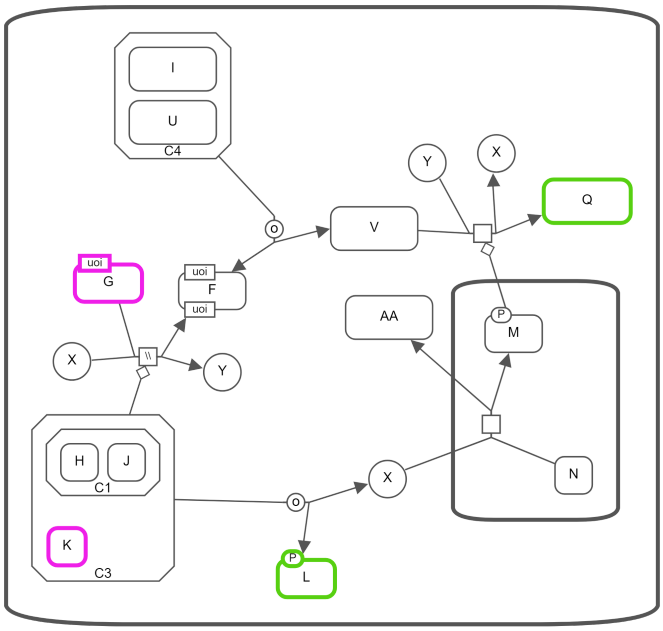
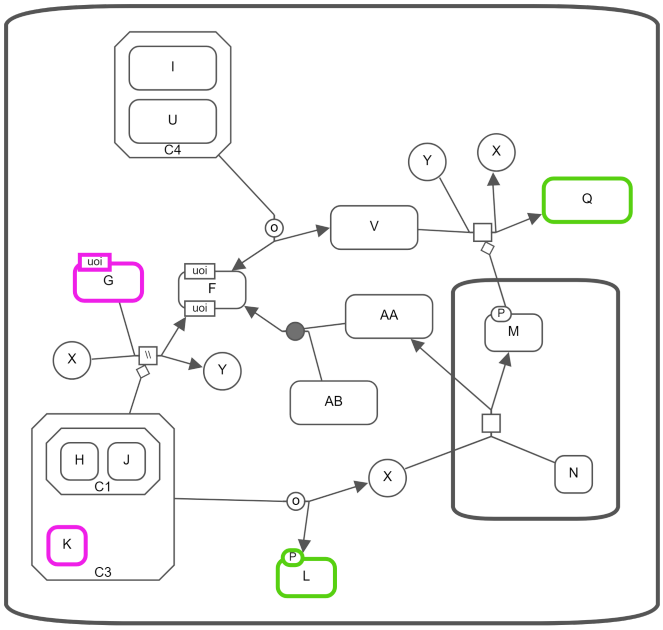


Figure 3.16: Paths-From-To query on the SBGN map in Figure 3.10. (a) Paths from G and K to L and Q with additional distance 0. (b) Paths from G and K to L and Q with additional distance 1.



(a)



(b)

Figure 3.17: Paths-From-To query on the SBGN map in Figure 3.10. (a) Paths from G and K to L and Q with additional distance 2. (b) Paths from G and K to L and Q with additional distance 3.

3.3 Experiments on Performance

We have performed experiments to measure performance of our graph based user defined procedures. In our first experiment, we measure the performance of our procedures on large graphs. For this reason, we use a large SBGN map which consists of 22706 nodes and 24628 edges. We select a sub-graph of this large SBGN map to create graphs with varying size and measure how the size affects the performance. By preserving the selected sub-graph, we remove remaining nodes and edges keeping the map intact with respect to SBGN notation to prepare SBGN maps with 4514, 8861, 13641, and 18154 nodes.

Graph based user defined procedures run on prepared SBGN maps of varying size. Results of procedure performances are shown in Figure 3.18 through Figure 3.21. Series on the charts are labeled with number of returned nodes. For example, in Figure 3.18, when we run *Neighbourhood* procedure with length limit of 1, it returns 142 nodes and blue series is labeled with 142, the execution taking 281 ms.

As it is shown in Figure 3.18 through Figure 3.21, run time complexity of our graph based queries seems to be linearly related to the number of nodes for different length limits.

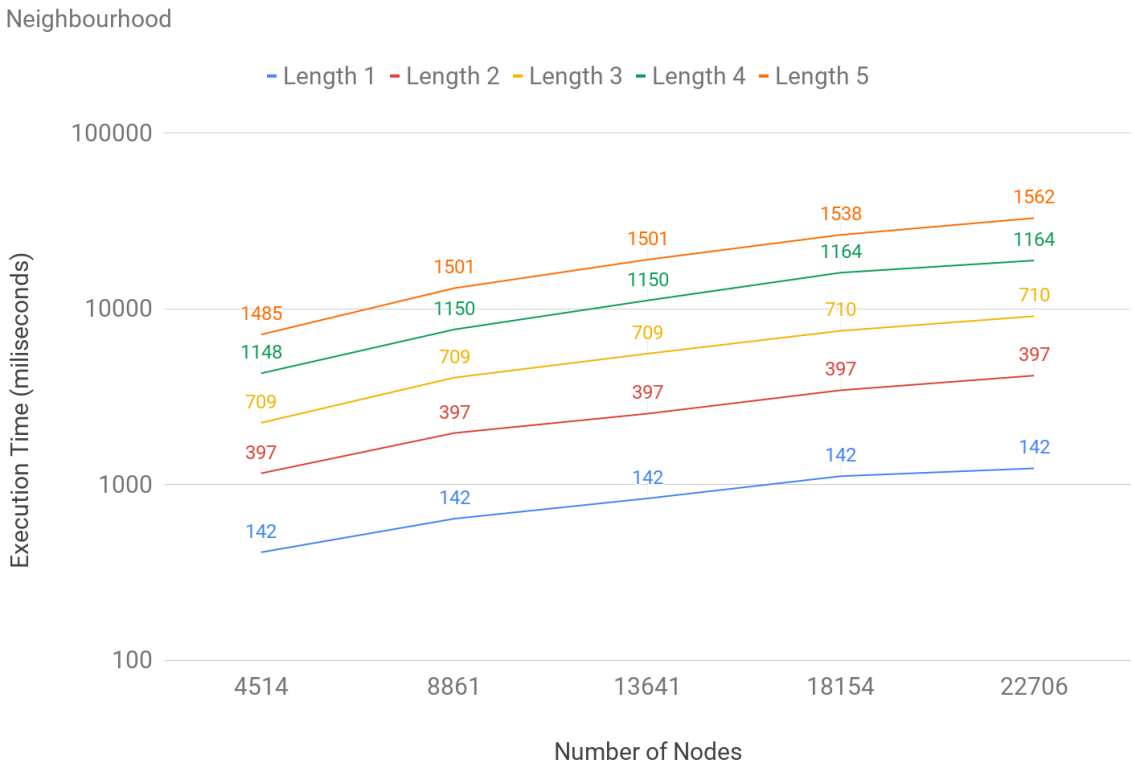


Figure 3.18: Neighbourhood Procedure - Execution Time vs Number of Nodes chart on large SBGN maps. Notice that the number of edges in returned graphs should be proportional to the number of nodes in SBGN maps.

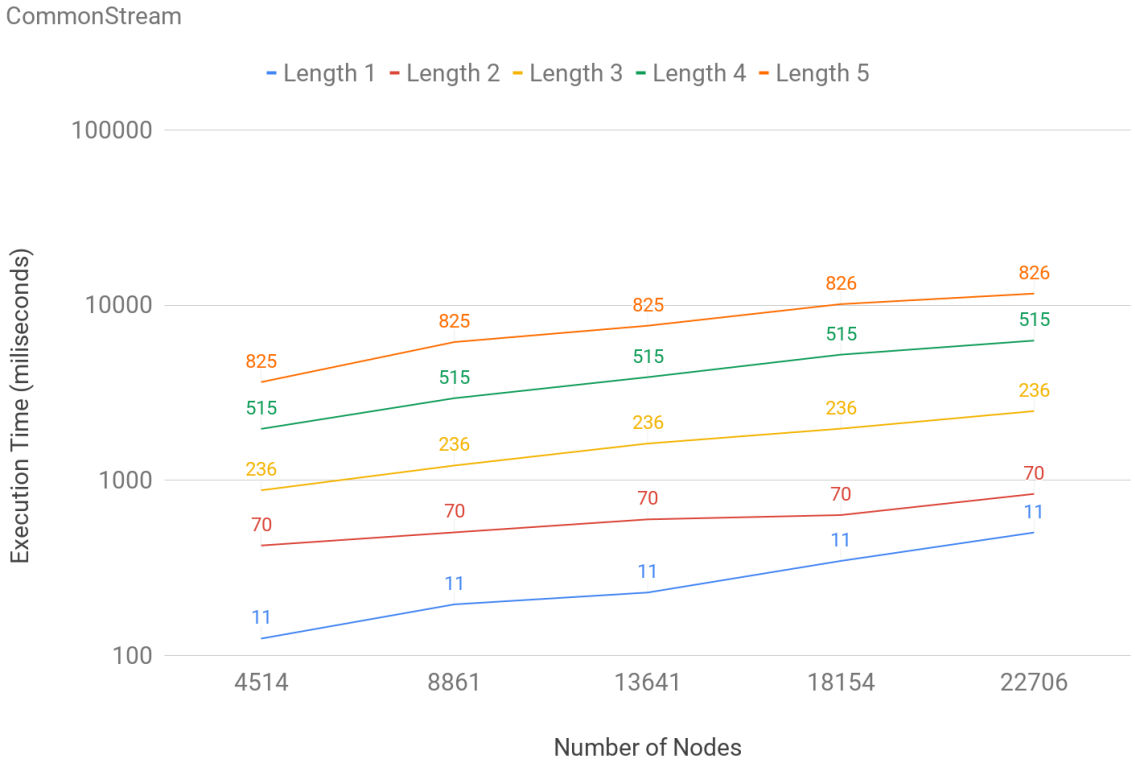


Figure 3.19: Common Stream Procedure - Execution Time vs Number of Nodes chart on large SBGN maps. Notice that the number of edges in returned graphs should be proportional to the number of nodes in SBGN maps.

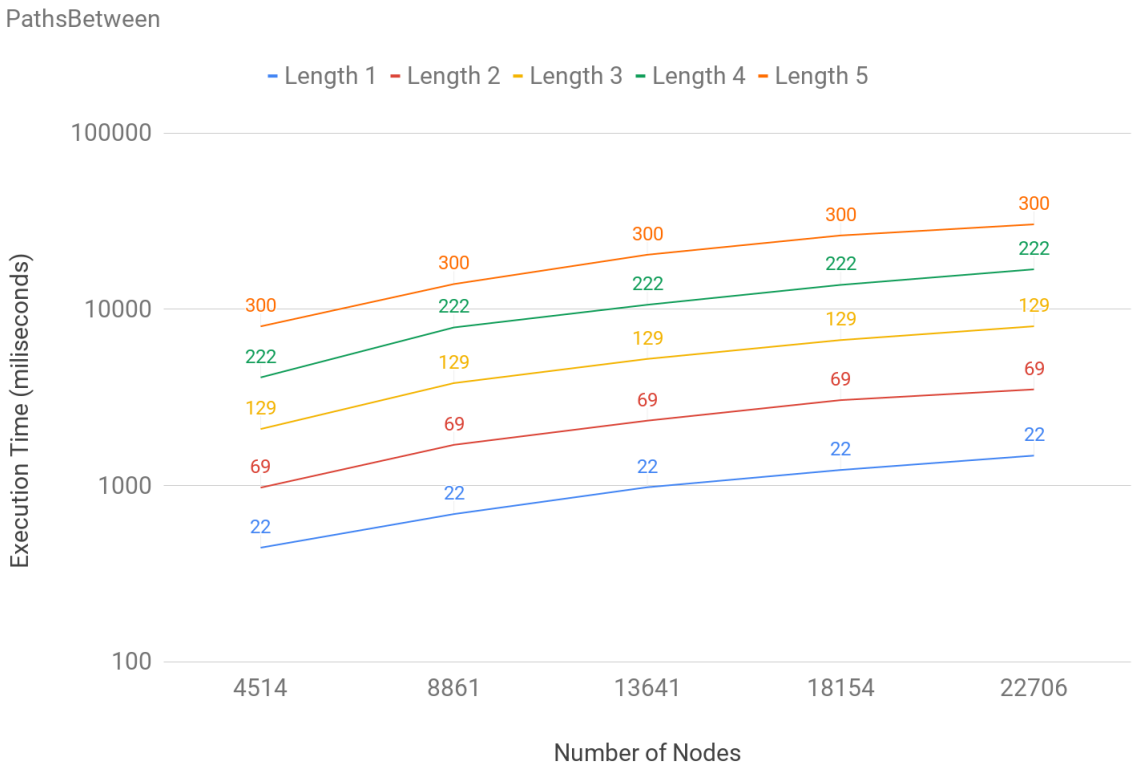


Figure 3.20: Paths-Between Procedure - Execution Time vs Number of Nodes chart on large SBGN maps. Notice that the number of edges in returned graphs should be proportional to the number of nodes in SBGN maps.

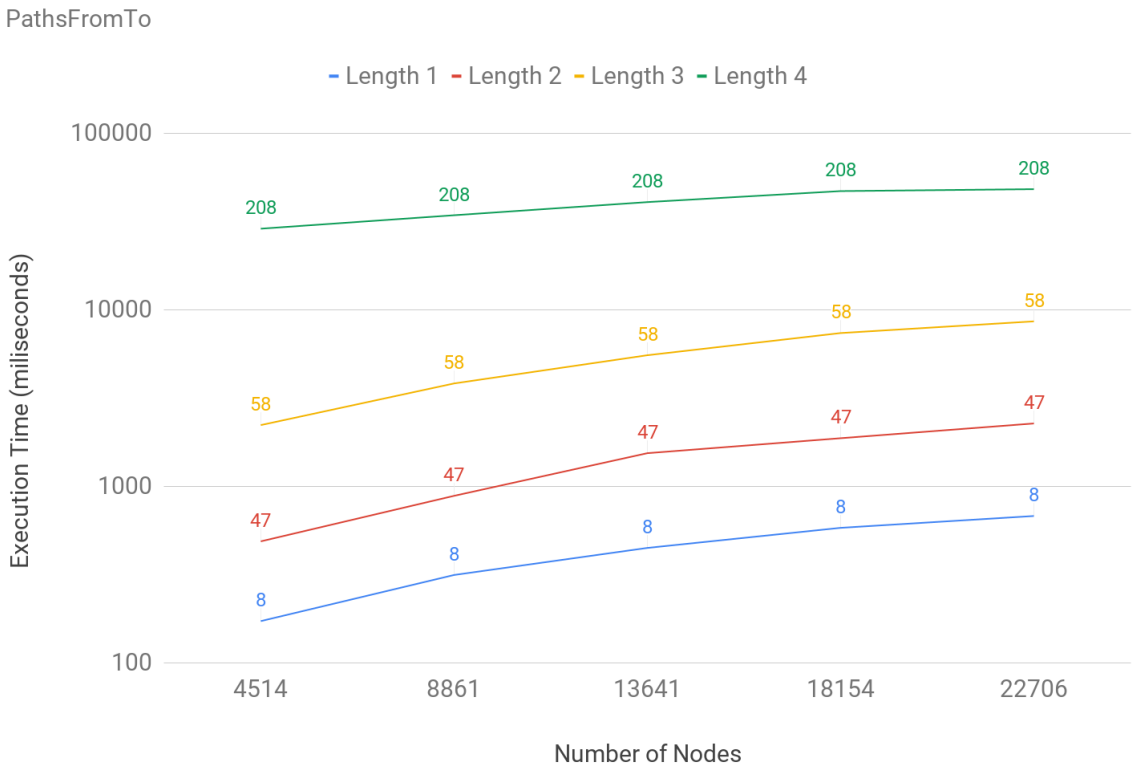


Figure 3.21: Paths-From-To Procedure - Execution Time vs Number of Nodes chart on large SBGN maps. Notice that the number of edges in returned graphs should be proportional to the number of nodes in SBGN maps.

Figure 3.22 confirms that run time complexity of *Paths-Between* procedure is linear in result size (number of nodes and edges returned by the query) as it is mentioned in [21]. Notice that the number of edges in returned graphs should be proportional to the number of nodes in SBGN maps. This applies to all query types.

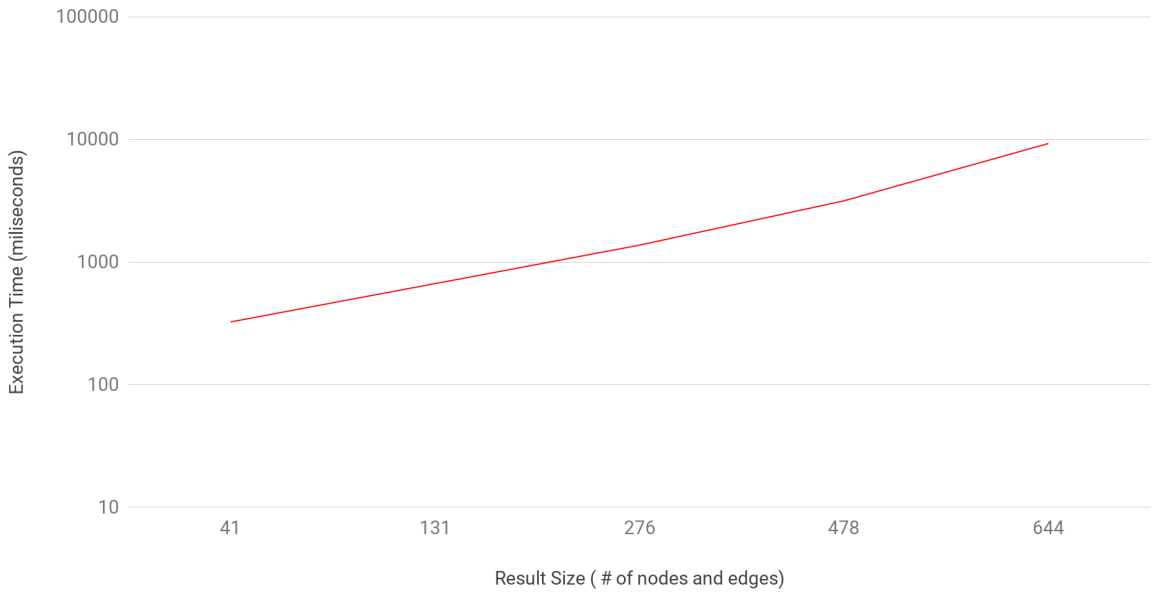


Figure 3.22: Paths-Between Procedure - Execution Time vs Result Size (number of nodes and edges) chart on SBGN map with 4514 nodes

Figure 3.23 implies that run time complexity of *Paths-From-To* procedure shows exponential increase in length limit. The simple reason for this is that rise in length limit increases total length of the paths enumerated exponentially due to the branching introduced with each node [21].

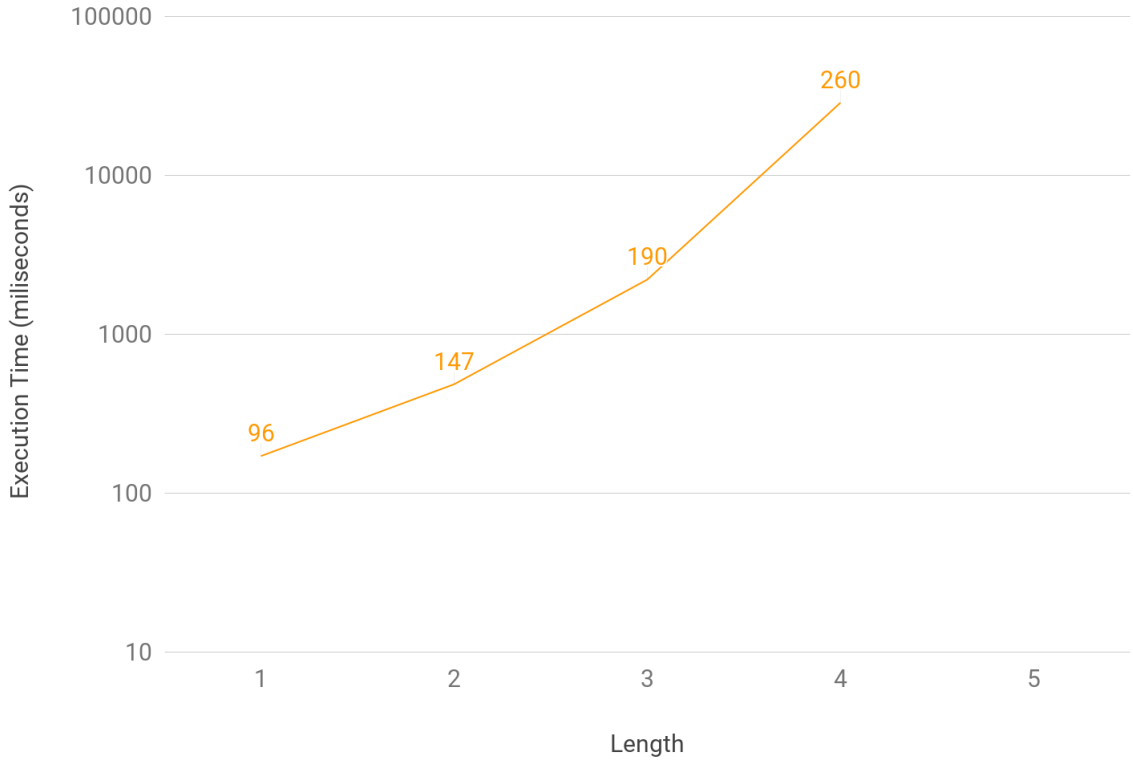


Figure 3.23: Paths-From-To Procedure - Execution Time vs Length chart on SBGN map with 4514 nodes

Furthermore, we ran our procedures on SBGN maps having smaller sizes. These SBGN maps have 1065, 1281 and 880 nodes, and proportional number of edges.

If we evaluate charts in Figures 3.24, 3.25 and 3.26, it is inferred that running graph based queries on the SBGN map with 1281 nodes does not perform well compare to SBGN maps with 1065 nodes and 880 nodes. The reason why procedures perform poorly on the SBGN map with 1281 nodes is that it contains more compound nodes and same entity occurs more frequently in the associated SBGN map.

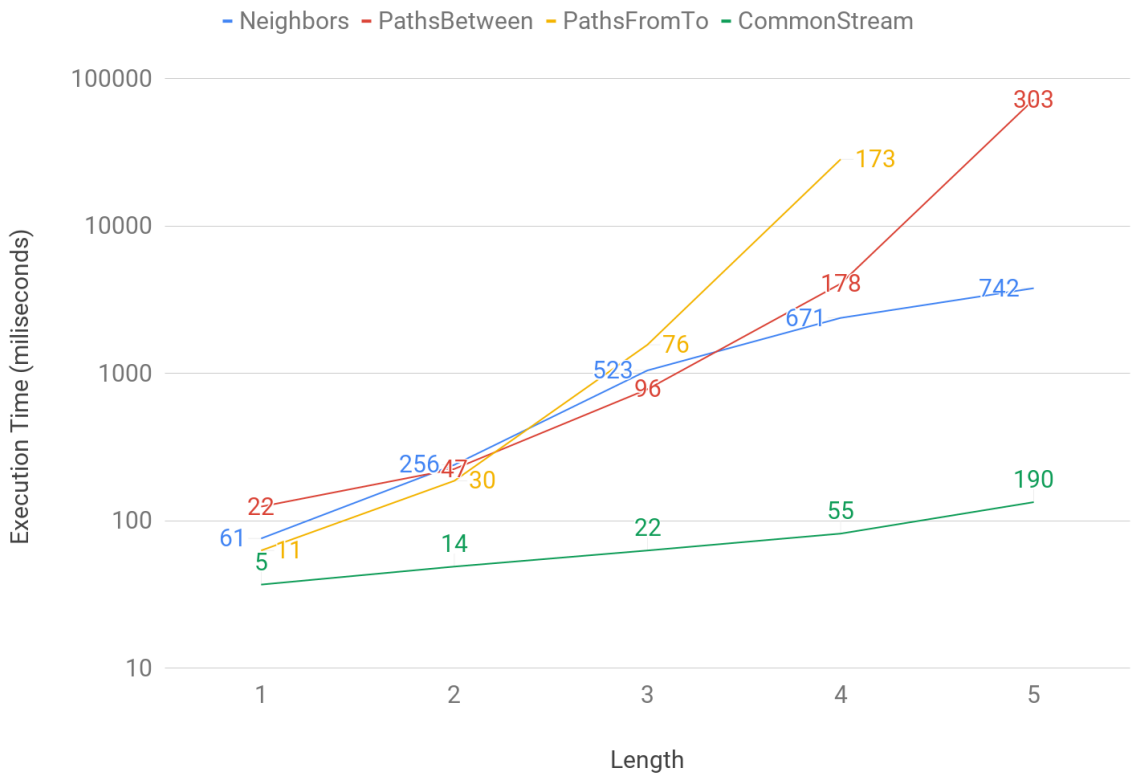


Figure 3.24: Execution Times of procedures on the SBGN map with 1065 nodes vs Length

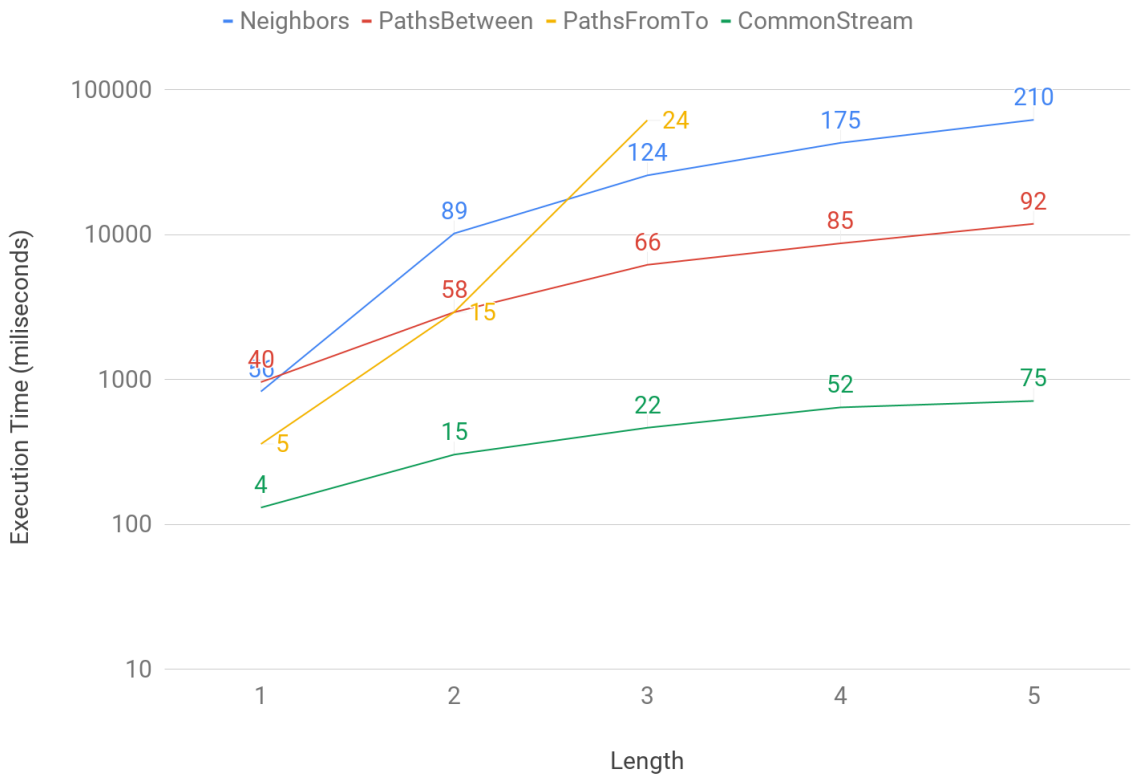


Figure 3.25: Execution Times of procedures on the SBGN map with 1281 nodes vs Length

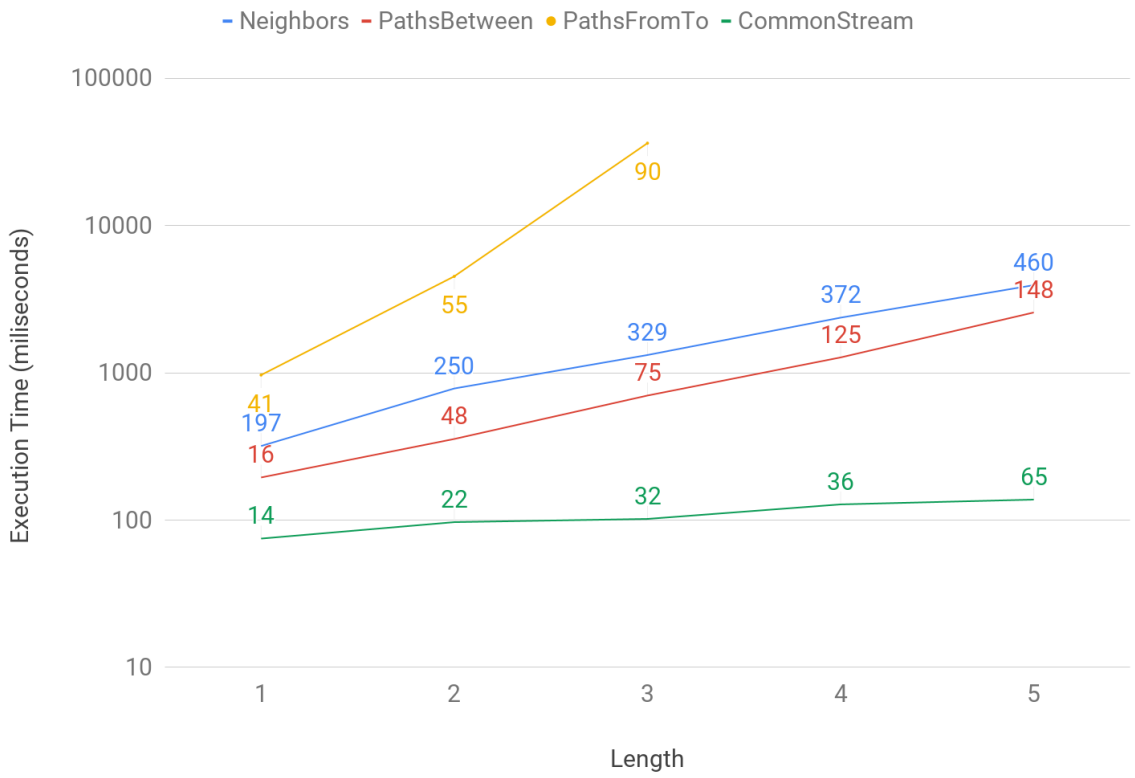


Figure 3.26: Execution Times of procedures on the SBGN map with 880 nodes vs Length

As a result, our experiments confirm the theoretical performance of the query algorithms in [21].

Chapter 4

Adding Local Database Support to Newt

Implemented user defined procedures with Cypher for graph algorithms were made available to call via Cypher statements. After procedures are deployed, database users of Neo4j can call our procedures with the Neo4j browser or Neo4j driver which is available for Java, .Net, JavaScript and Python [22]. In fact, these user defined procedures can be used by any SBGN related tool or software with support for loading or creating SBGN-ML text. Anyone who wants to insert/read SBGN maps or has an interest to our graph algorithms can call our procedures by using the Neo4j driver within their SBGN related tool.

Newt is an SBGN related visualization tool that can produce or consume SBGN-ML text and Newt is based on ChiSE.js extending SBGNViz.js provides loadSBGNMLText() method for loading graph from SBGN-ML text and createSbgnml() method for creating SBGN-ML text for current SBGN map. Therefore, the Newt editor can be integrated to Neo4j to call our user defined procedures. Since Newt editor is implemented in JavaScript language, Neo4j's official JavaScript driver was added to Newt as a dependency to properly integrate it with the Neo4j graph database.

Architecture of integration of Newt and local Neo4j graph database is presented in Figure 4.1. In order to connect Newt editor to Neo4j graph database, Newt uses a web service that utilizes Node.js [23].

In order to call each implemented user defined procedure from Newt, we implemented methods for the web services to execute Cypher statements with ajax requests. Implemented methods of web service take two arguments which are *request* and *respond*. Request argument is responsible for containing parameters which are passed from the ajax call. By request arguments, Cypher statement which calls the user defined procedure is executed. After Cypher statement is executed, the result is sent back to where it makes an ajax call with the respond argument.

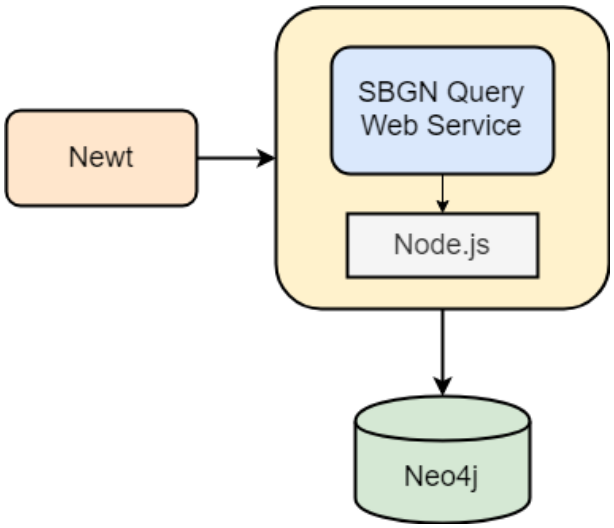


Figure 4.1: Architecture of Newt and Neo4j Integration

In the rest of this chapter, you will find the operations, with their graphical user interface, related to the local database support in Newt.

4.1 Save to Graph DB

To call *Insert Graph* on Newt UI, we add a drop-down menu item under the File menu of Newt (Figure 4.2). When clicking on *Save to Graph DB*, it will fire an event to call a function to create an SBGN-ML text of the current map in the Newt editor by *createSbgnml()* method of SBGNViz.js. After the SBGN-ML text is created, it is given as a parameter to ajax post request to invoke the web service. Implemented web service method gets SBGN-ML text from the request argument to execute Cypher statement to call the *Insert Graph* procedure.

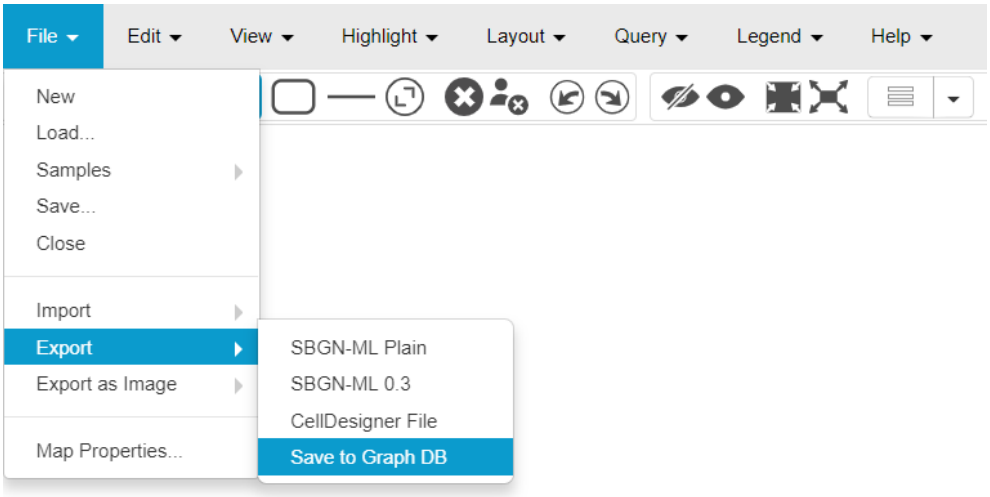


Figure 4.2: Menu item to save graph to database

4.2 Read From Graph DB

In order to invoke *Read From Graph DB* procedure from the user interface of the Newt editor, a new item is added to its menu (Figure 4.3). Clicking on *Read from Graph DB* triggers a function call that makes an ajax get request to invoke a web service. Then, web service returning SBGN-ML text from execution of Cypher statement for calling *Read From Graph DB* procedure. Then, the SBGN map is loaded to Newt by calling SBGNViz's *loadSBGNMLText()* method with returned

SBGN-ML text.

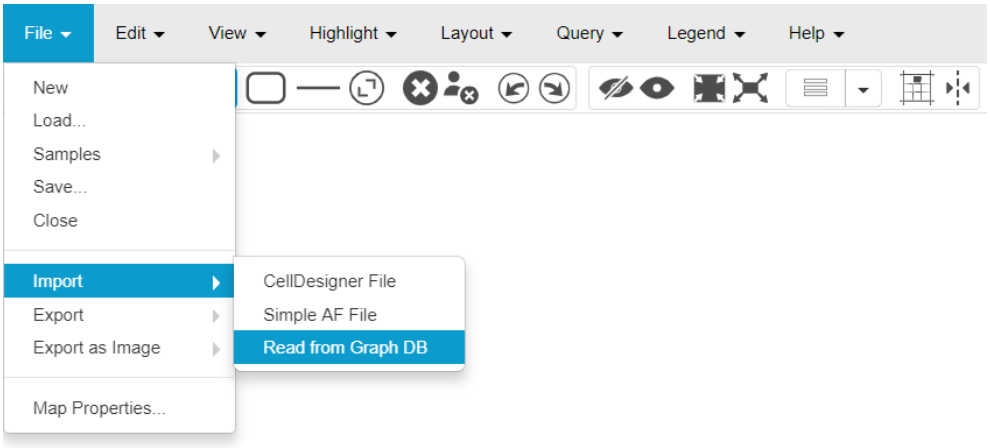
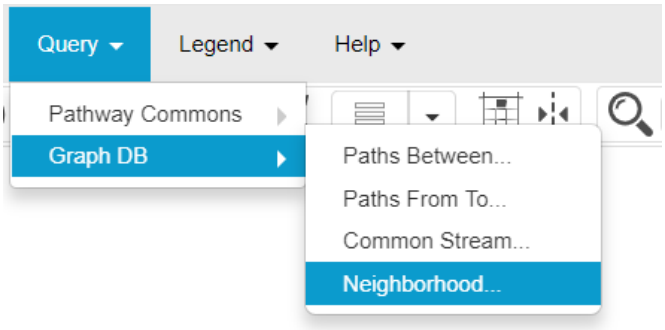


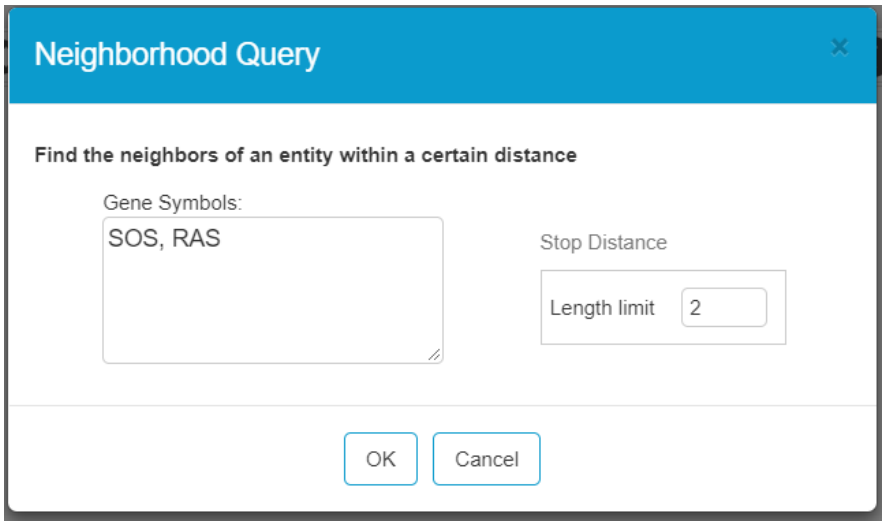
Figure 4.3: Menu item to read from the graph database

4.3 Neighbourhood Query

For calling the *Neighbourhood* procedure, a menu item is added to menu of Newt (Figure 4.4a). Clicking on *Neighbourhood..* on under the Query menu opens Neighbourhood Query dialog which has inputs for gene symbols and length limit (Figure 4.4b). Clicking on OK button, after filling input fields with desired genes and length limit for stop distance performs an ajax get request to web service method. The web service executes the *Neighbourhood* procedure with genes list and length limit to return resulting sub-graph as SBGN-ML text. After SBGN-ML text is returned from the ajax request, sub-graph is loaded by *loadSBGN-MLText()* method of SBGNViz.js to visualize returned sub-graph in the Newt editor.



(a)



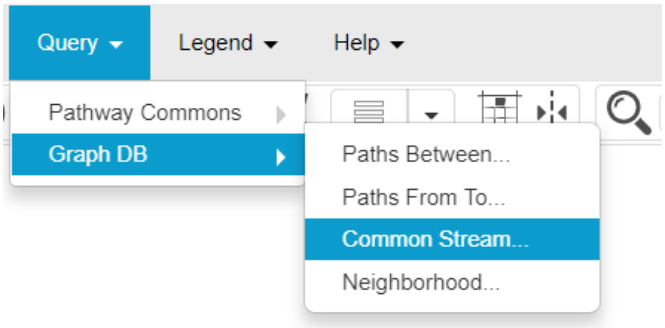
(b)

Figure 4.4: (a) Menu item for Neighbourhood query. (b) Neighbourhood query dialog of Newt.

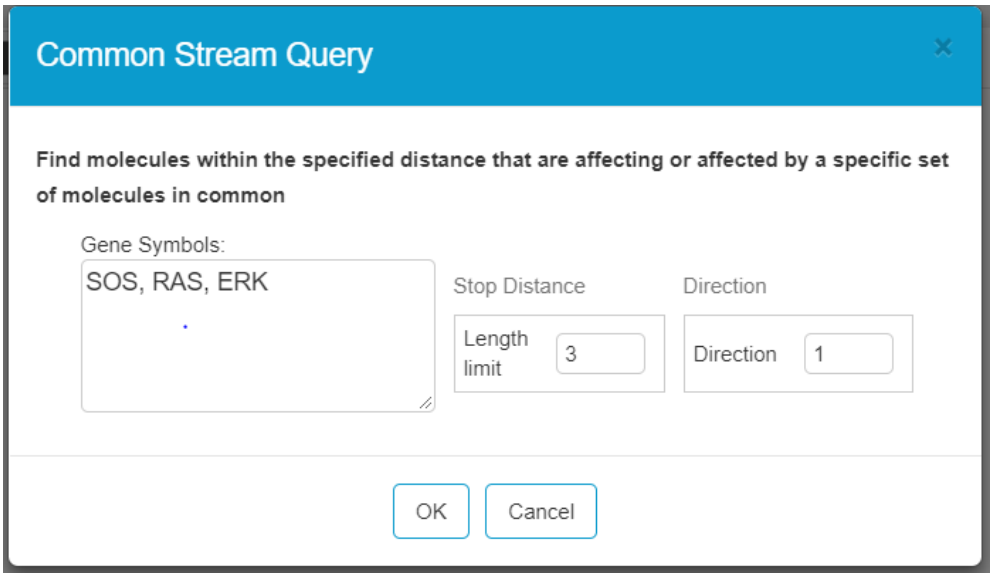
4.4 Common Stream Query

In order to call *Common Stream* procedure on Newt, a new Query menu item was added to Newt to show Common Stream Query dialog which takes a gene list, direction and length limit (Figure 4.5). After this dialog is filled out, a web service method which executes *Common Stream* procedure is called via an ajax request. This ajax request returns SBGN-ML text to Newt for loading an SBGN

map by the *loadSBGNMLText()* method.



(a)



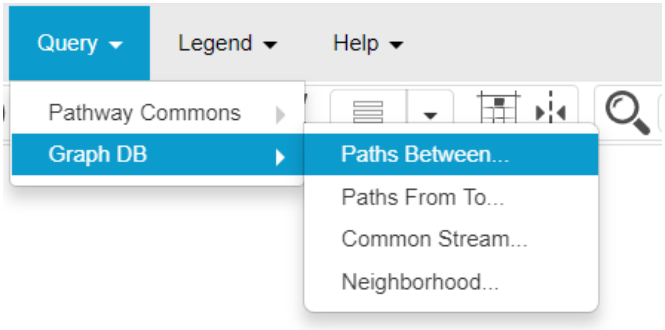
(b)

Figure 4.5: (a) Menu item for Common Stream query. (b) Common Stream query dialog of Newt.

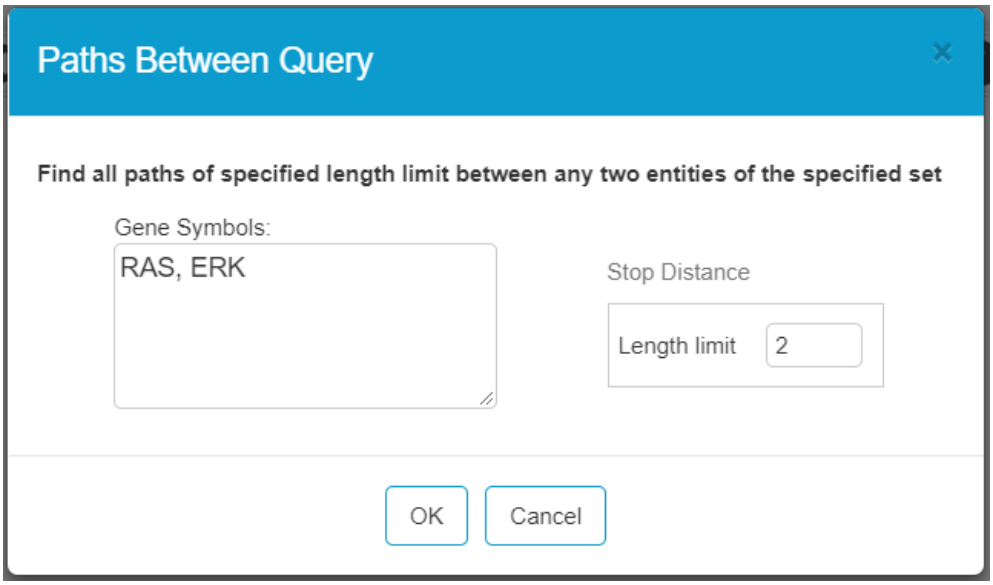
4.5 Paths-Between Query

We add a menu item to Newt in order to open Paths-Between query dialog to get parameters from user interface to call a web service method. Then, the web service performs an ajax request for executing Cypher statement to call *Paths-Between* procedure with given parameters in Figure 4.6. Returned SBGN-ML

text from the web service is given to the *loadSBGNMLText()* of SBGNViz.js for loading SBGN maps to Newt.



(a)



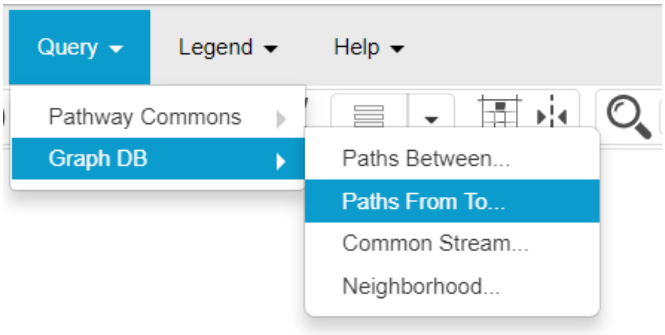
(b)

Figure 4.6: (a) Menu item for Paths-Between query. (b) Paths-Between query dialog of Newt.

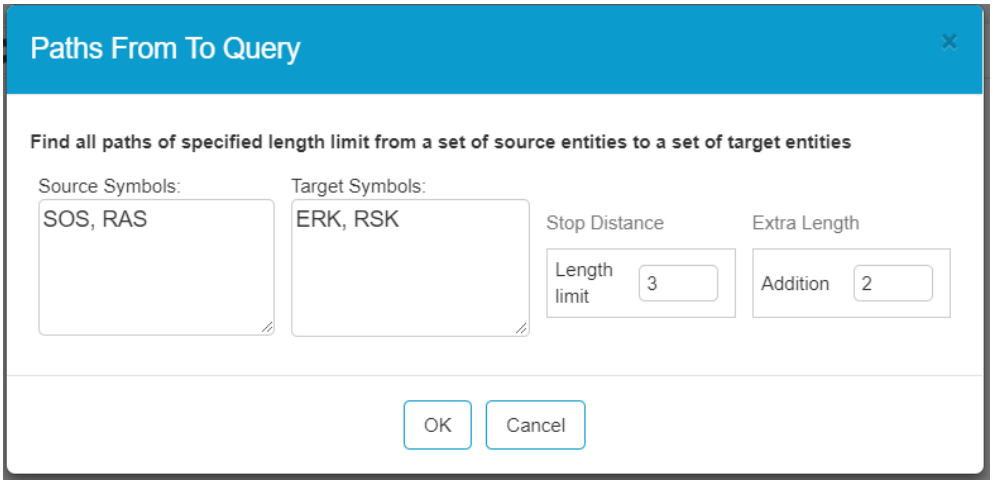
4.6 Paths-From-To Query

To call *Paths-From-To* procedure via Newt, another menu item was added to the user interface of Newt for opening Paths-From-To query dialog (Figure 4.7). Filling query editor with target genes, source genes, limit and additional distance

parameters and clicking on OK button makes a ajax call to a web service method. Then, the web service executes a Cypher statement which calls *Paths-From-To* procedure and returns the result as SBGN-ML text. Like other graph based user defined procedures, *loadSBGNMLText()* is used for visualizing SBGN-ML text.



(a)



(b)

Figure 4.7: (a) Drop-down menu item for Paths-From-To query. (b) Paths-From-To query dialog of Newt.

Chapter 5

Conclusion

In this thesis, we developed a representation in Neo4j for SBGN maps. We also implemented graph based user defined procedures and queries using these procedures as a Neo4j plugin.

We model SBGN maps in Neo4j graph database by respectively mapping arc, glyph and attributes of SBGN map to node, relationship and property of property graph data model. In order to model compound structures, we create a *resideIn* relationship between parent node and children nodes.

We designed a breadth first traversal method which handles compound structures by using Cypher statements consisting defined pattern with newly defined *resideIn* relationship. Moreover, compound breadth first traversal method respects our definition of length. It accepts length of paths, which goes from entity node to a process node and then goes from the process node to another entity node, as 1 despite the fact that count of relationships in this path is 2 to properly take the bipartite nature of the underlying graph into account.

By using the mentioned data model and compound based breadth first traversal method, we implement graph based user defined procedures such as *Neighbourhood*, *Common Stream*, *Paths-Between* and *Paths-From-To* that take/return SBGN-ML text.

Furthermore, a web service was implemented to integrate Newt with Neo4j so that implemented user defined procedures can be executed via the graphical user interface of Newt.

5.1 Future Work

In current situation, while inserting a new SBGN map to graph database, we delete any stored graph from the database. The reason is that we can not merge SBGN maps yet. Therefore, as a future work, we are planning to merge SBGN maps during inserting SBGN maps to the graph database. However, merging SBGN maps is not straightforward like it is addressed in STON as it requires matching different states of macromolecules, which are often expressed differently (e.g. phosphorylated state of p53 vs. p53 phosphorylated at serine 389). One idea here might be to do this semi-automatically, where we involve the user in making such matching decisions.

Bibliography

[1] S. Moodie, N. Le Novere, E. Demir, H. Mi, and A. Villeger, "Systems biology graphical notation: process description language level 1 version 1.3," *Journal of integrative bioinformatics*, vol. 12, no. 2, pp. 213–280, 2015.

[2] "How neo4j co-exists with oracle rdbms: An introduction to graphs." https://neo4j.com/blog/neo4j-co-exists-oracle-rdbms-introduction-graphs/. Accessed: 2018-12-25.

[3] "User defined procedures and functions." https://neo4j.com/developer/procedures-functions/. Accessed: 2018-12-25.

[4] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader, "Cytoscape.js: a graph theory library for visualisation and analysis," *Bioinformatics*, vol. 32, no. 2, pp. 309–311, 2016.

[5] "Newteditor." http://newteditor.org/. Accessed: 2018-12-25.

[6] N. Le Novere, M. Hucka, H. Mi, S. Moodie, F. Schreiber, A. Sorokin, E. Demir, K. Wegner, M. I. Aladjem, S. M. Wimalaratne, *et al.*, "The systems biology graphical notation," *Nature biotechnology*, vol. 27, no. 8, p. 735, 2009.

[7] M. P. Van Iersel, A. C. Villéger, T. Czauderna, S. E. Boyd, F. T. Bergmann, A. Luna, E. Demir, A. Sorokin, U. Dogrusoz, Y. Matsuoka, *et al.*, "Software support for sbgn maps: Sbgn-ml and libsbgn," *Bioinformatics*, vol. 28, no. 15, pp. 2016–2021, 2012.

[8] "Procedures." https://neo4j.com/docs/java-reference/current/extending-neo4j/procedures/.

[9] "Concepts: Relational to graph." https://neo4j.com/developer/graph-db-vs-rdbms/.

[10] V. Touré, A. Mazein, D. Waltemath, I. Balaur (Roznovat, M. Saqi, R. Henkel, J. Pellet, and C. Auffray, "Ston: Exploring biological pathways using the sbgn standard and graph databases," *BMC Bioinformatics*, vol. 17, 12 2016.

[11] M. Siper, "Libraries and tools for viewing and editing biological maps in sbgn," Master's thesis, Bilkent University, Bilkent University, 2017. Advisor: Doğrusöz, Uğur.

[12] A. Lysenko, I. Balaur (Roznovat, M. Saqi, A. Mazein, C. Rawlings, and C. Auffray, "Representing and querying disease networks using graph databases," *BioData Mining*, vol. 9, 12 2016.

[13] "Properties: The heart of graph databases." https://neo4j.com/business-edge/properties-the-heart-of-graph-databases/. Accessed: 2018-12-25.

[14] "Graph database concepts." https://neo4j.com/docs/developer-manual/current/introduction/graphdb-concepts/. Accessed: 2018-12-25.

[15] "What is a graph database?." https://neo4j.com/developer/graph-database/. Accessed: 2018-12-25.

[16] "Cypher query language." https://neo4j.com/developer/cypher/.

[17] "Cypher basics 1." https://neo4j.com/developer/cypher-query-language/.

[18] "Neo4j java driver." https://mvnrepository.com/artifact/org.neo4j.driver/neo4j-java-driver. Accessed: 2018-12-25.

[19] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir, "A layout algorithm for undirected compound graphs," *Inf. Sci.*, vol. 179, pp. 980–994, Mar. 2009.

[20] U. Dogrusoz, A. Karacelik, I. Safarli, H. Balci, L. Dervishi, and M. C. Siper, "Efficient methods and readily customizable libraries for managing complexity of large networks," *PLOS ONE*, vol. 13, pp. 1–18, 05 2018.

[21] U. Dogrusoz, A. Cetintas, E. Demir, and O. Babur, "Algorithms for effective querying of compound graph-based pathway databases," *BMC bioinformatics*, vol. 10, p. 376, 11 2009.

[22] "Language guides." https://neo4j.com/developer/language-guides/. Accessed: 2018-12-25.

[23] "Node.js." https://nodejs.org/en/. Accessed: 2018-12-25.