**Open University of Israel**

The Department of Computer Science

**Advanced Project in Computer Science**



**Part of M.A Degree in CS**

Research Supervisor:

**Prof. Ehud Gudes**

By Mr. Yitzhak Orfanian

ID: 031860984

Email: [iziaman@gmail.com](mailto:iziaman@gmail.com)

Table of Contents

[Acknowledgments 4](#_Toc96802227)

[**1.** **Introduction** About MGO – Medical Graph Ontology 4](#_Toc96802228)

[**2.** **The main components** 8](#_Toc96802229)

[Graph Database 8](#_Toc96802230)

[Can Every Database Represent a Graph? 8](#_Toc96802231)

[MongoDB document database 8](#_Toc96802232)

[*MongoDB Storage* [2] 8](#_Toc96802233)

[MongoDB Queries 8](#_Toc96802234)

[How to Run Queries 9](#_Toc96802235)

[Neo4j Graph Data Platform. 9](#_Toc96802236)

[Neo4j DBMS server 10](#_Toc96802237)

[Access Neo4j DBMS 10](#_Toc96802238)

[Cypher Language 11](#_Toc96802239)

[Node JS & Express 12](#_Toc96802240)

[Node JS 12](#_Toc96802241)

[NPM 12](#_Toc96802242)

[Node Modules 13](#_Toc96802243)

[MVC – Model View Controller 14](#_Toc96802244)

[Model 14](#_Toc96802245)

[View 14](#_Toc96802246)

[Controller 14](#_Toc96802247)

[**3.** MGO – Medical Graph Ontology Overview. 16](#_Toc96802248)

[MGO Key Features 16](#_Toc96802249)

[MGO Project Functionality 17](#_Toc96802250)

[User Control 17](#_Toc96802251)

[Symptom Manager 17](#_Toc96802252)

[Treatment Manager 17](#_Toc96802253)

[Tags Manager 17](#_Toc96802254)

[Add bulk data 17](#_Toc96802255)

[Analyze Data 18](#_Toc96802256)

[MGO User Guide 18](#_Toc96802257)

[Main Page 18](#_Toc96802258)

[User Control 18](#_Toc96802259)

[Symptoms 20](#_Toc96802260)

[Treatments 22](#_Toc96802261)

[Tags 24](#_Toc96802262)

[Bulk Actions 25](#_Toc96802263)

[Analyze Data 26](#_Toc96802264)

[Union Search 26](#_Toc96802265)

[All Shorts Path 27](#_Toc96802266)

[Cypher Queries 29](#_Toc96802267)

[**4.** The Code 30](#_Toc96802268)

[*The Directory Structure*: 30](#_Toc96802269)

[**5.** **Summary and Future Development** 30](#_Toc96802270)

[**Appendix A – The software main modules** 31](#_Toc96802271)

[The Main Files 31](#_Toc96802272)

[./package.JSON 31](#_Toc96802273)

[./app.js 31](#_Toc96802274)

[./config/passport.js 32](#_Toc96802275)

[./config/auth 32](#_Toc96802276)

[./db/neo\_connection.js 32](#_Toc96802277)

[./models/User.js 32](#_Toc96802278)

[./models/Queries.js 32](#_Toc96802279)

[./models/BaseNode.js 33](#_Toc96802280)

[class Data{..} 34](#_Toc96802281)

[./models/Tag.js 35](#_Toc96802282)

[./routes directory 35](#_Toc96802283)

[./controller directory 35](#_Toc96802284)

[./controller/bulk.js 36](#_Toc96802285)

[./controller/analyze.js 36](#_Toc96802286)

[**Appendix B – Review of main concepts** 38](#_Toc96802287)

[Ontology 38](#_Toc96802288)

[Ontology Usage 39](#_Toc96802289)

[SPARQL 39](#_Toc96802290)

[Resource Description Framework 40](#_Toc96802291)

[Protégé 41](#_Toc96802292)

[Class expressions 42](#_Toc96802293)

[Graph Theory 43](#_Toc96802294)

[Graph Categories: 44](#_Toc96802295)

[Database 44](#_Toc96802296)

[Query Language (QL) 45](#_Toc96802297)

[Database Types 45](#_Toc96802298)

[Neo4j Graph Database 45](#_Toc96802299)

[Neo4j Storage 45](#_Toc96802300)

[Neo4j Licenses Type 46](#_Toc96802301)

[Figures Table 47](#_Toc96802302)

[Bibliography 48](#_Toc96802303)

# Acknowledgments

The project is part of the MA in Computer Science at the Open University of Israel.

I would like to thank my mentor, Prof. Ehud Gudes. He guided me, helped me to understand the main goal, directed me to the best technologies to achieve my goals, and help understand my needs.

The family that gave me the space to exposure and learn developed and guided me a lot. Especially my wife Calanit who I nagged. I talked about every minor achievement I passed, and she always gave me the space and guided me.

This project is in memory of my mother, Sara Orfanian, who taught me not to give up no matter how hard the challenge.

# **Introduction** About MGO – Medical Graph Ontology

This project is part of my CS studies.

I have had a lot of medical issues in my life.

I have received a lot of treatments for my symptoms but have always wondered if I had received alternative treatments, would they have made me feel better. Maybe they would have resolved many of the symptoms I still suffer from today.

With the advice of Prof. Ehud Gudes, we set the project goal to create a medical ontology of medical issues. However, the first association I had was a graph data structure that needs to hold the data (I will elaborate on these terms later in the article).

After long research, I found that most ontology infrastructures keep the data in flat XML files and the data association is logical. I found that if the project uses a graph database, the project will have graph algorithm abilities and better performance to analyze the data. There are several graph DB types available, I chose the Neo4j Graph database for reasons that I will explain in this article.

A picture containing text, businesscard

Description automatically generated

Figure 1 - Project components flow

The project infrastructure contains the Neo4j for holding the symptoms, treatments, and relative data, MongoDB for user authentication, Node JS and Node JS modules. In figure 1, we can see a simple diagram of the relations between a component in the project.

MGO Key Features:

* Medical ontology of symptoms, treatments, and their relations.
* Friendly User Interface:
  + Display, add, and analyze data.
* Run Neo4j cypher queries from the portal.
* Free to use under GNU GPLv3 license.

I hope the project will help discover and understand any unknown issues from different treatments.

Figure 2 shows the main window of the MGO project.

Article Coverage.

In this article we will cover the following issues:

* Overview of the databases I used in the project:
  + MongoDB.
  + Neo4j and the Cypher language.
* Overview of Node JS and the main modules I used in the project.
* MGO – user guide.

In the appendix, I have included some relevant information that I researched:

* Ontology and existing tools:
  + SPARQL and RDF
  + Protégé
* Graph theory.
* Database.
* More information on Neo4j.

Graphical user interface

Description automatically generated

Figure 2- MGO main window

# **The main components**

## Graph Database

A database that uses graph data structure and semantic queries that use nodes and edges query. Most graph databases can run graph algorithms. Graph databases use different query languages that match the vendor. The known languages are Cypher and the top vendors are, Neo4j, Hadoop Graph, and JanusGraph.

### Can Every Database Represent a Graph?

We can represent a graph using different database types but using the Neo4j mechanism give us advantages in:

* Entering data into the database is much quicker, and less unneeded information is entered.
* Minimize the number of queries and their complexity to get the same results.

## MongoDB document database

MongoDB [1] is a source-available, cross-platform, document-oriented database that uses JSON documents with optional schema.

It is defined as a NoSQL database program, and by that, the DBMS queries the documents by keys. As defined before, not all documents must contain duplicate keys (but it is recommended).

MongoDB is licensed under the Server Side Public License (SSPL).

### MongoDB Storage [2]

The storage engine is the primary component of MongoDB responsible for managing data. It manages how data is stored in memory and on disk. MongoDB supports multiple storage engines.

MongoDB stores the data on BSON files. Usually, it will hold two files per collection, collection.0, which stores the data, and collection. ns, which stores the namespacing metadata for the collection.

### MongoDB Queries

The DBMS holds collections, which in turn hold JSON documents.There are many available queries, for example, to get the first document by name:

db.mycol.findOne({title: "MongoDB Overview"}), this will return the first document that his title is “MongoDB Overview”.

### How to Run Queries

The main ways to enter data to MongoDB are:

* Using MongoDB console "mongod".
* Using MongoDB GUI client Robo3T.
* Using MongoDB driver for different programming languages.
* Using the object modeling tool (Mongoose) lets you create an object representing the collection schema and run crud on the object to update the records.

## Neo4j Graph Data Platform.

Neo4j [3] is a graph database management system developed by Neo4j, Inc. Neo4j is available in a GPL3-licensed (free to run) open-source "community edition".

The key features are:

* SQL Like easy query language Neo4j CQL.
* Follows Property Graph Data Model.
* Supports Indexes by using Apache Lucence.
* Supports UNIQUE constraints.
* Contains a UI to execute CQL Commands in Neo4j Data Browser.
* Supports complete ACID (Atomicity, Consistency, Isolation, and Durability) rules.
* Uses Native graph storage with Native GPE (Graph Processing Engine).
* Supports exporting of query data to JSON and XLS format.
* Provides REST API to be accessed by any Programming Language like Java, Spring, Scala, etc.
* Provides JavaScript to be accessed by any UI MVC Framework like Node JS.
* Supports two kinds of Java API: Cypher API and Native Java API to develop Java applications.

The most critical issue for our project is the visualization tools that represent the Graph. With Neo4j management, we can view the graph diagram. This tool was outsourced as a "neovis.js" javascript project. This can display the Graph on an HTML site.

### Neo4j DBMS server

We can run the Neo4j server database in several ways:

* On your personal computer by installing the Neo4j desktop application.
* As a docker container from the docker hub.
* Cloud server on most of the cloud vendors (AWS, Azure, GCP).
* Install the Neo4j server on a dedicated server.

### Access Neo4j DBMS

There are several ways to access the Neo4j DBMS and to manipulate the data:

* Neo4j Desktop – using the Neo4j browser, you can enter a Cypher query, and the output is displayed as a graph, table or JSON.
* Neo4j Console – entering Cypher queries and the output display as JSON or a table.
* Using the Neo4j driver for different programming languages like Java, C#, Python, JavaScript (Node JS), and more.

In figure 3, we can see the Neo4j desktop result window. It is similar to the MGO result window in figure 3.

[More about Neo4j at the appendix](#_Neo4J_Graph_Database)

Chart, bubble chart

Description automatically generated

Figure 3-Neo4j desktop window

### Cypher Language

Neo4j uses Cypher query language to store and retrieve data on Neo4j DBMS. It is an open-source part of the openCypher project. We can run a Cypher query from the Neo4j desktop or the programming driver.

#### Cypher Syntax

Cypher Language syntax provides a visual and logical way to match patterns of nodes and relationships in ASCII-Art syntax. For Example:

**MATCH(a:Treatment {name:$tretParam}),(b:Symptom {name:$sympParam})**

**MERGE(a)-[r:Side\_effect]->(b) RETURN a,b**

This query takes 2 nodes, Treatment and Symptom, that identify by name parameter and connect them (MERGE) using Side effect relations at the end of the operation, it returns the nodes.

**MATCH (synp:Symptom {name:"headache"})-[\*1..4]-(treatments)**

**RETURN DISTINCT treatments**

This query takes a symptom and returns treatments that are 4 hops away from this treatment.

We can see in the example the ASCII-Art of the arrow that represents the relation between the 2 nodes.

**Part of MGO analysis ability is to send a Cypher query to get a remarkable result.**

#### Cypher Results

If we run a Cypher query, it will return a JSON object. This object includes the following information:

Server version, Server address, the query we run, a summary that returns statistics on the query impacts, and the query results.

For every system, we will use the specific data that we need. Some systems like "Neovis" use the results object to display the graph data.

## Node JS & Express

For the project, I chose the Node JS programing language.

### Node JS

Open-source, cross-platform Node JS operates on the V8 engine and performs JavaScript code outside of a web browser to run JavaScript code. [4, 4]. Node JS gives the developer the ability to develop full-stack projects using JavaScript. Node JS is one of the most significant growing open source projects. Node JS key features:

* Non-blocking I/O.
* Single-thread event loop.
* Uses V8 execution engine.
* WebAssembly - WebAssembly (sometimes abbreviated Wasm) is an open standard that defines a portable binary-code format for executable programs, a corresponding textual assembly language, and interfaces for facilitating interactions between such programs and their programs host environment.

#### Running Node JS Program

Node JS is based on the existing Javascript programing language. To run any Node JS Program, you need to install a Node JS interpreter. The Node package includes another important program called NPM.

After installation, the server can be run in several ways:

* Node <serverscript>.js
* NPM start/run/debug if it defines in the package.json

### NPM

Npm [5] is the default package manager for the JavaScript runtime environment Node.JS. It consists of a command-line client called npm, and an online database of public and paid-for private packages called the npm registry.

Since it is very stable, many vendors use it to install programs and all their dependencies.

### Node Modules

Node JS uses a collection of "modules" that handle various core functionalities. These modules are like an extension to the Node JS core. This extension adds a new ability for the programmer to use. However, we can write these features we prefer to test and update code in our program and work on our goal.

To install a new module, we use the npm package manager

* npm install <packagename> --save

#### ECMAScript / ES6

ES6, also known as ECMAScript 2015, is the latest version of the ECMAScript standard. It is a revision of Javascript that adds new features for better writing and performance of the Javascript engine. The arrow function, "let" variable definition, defaults parameters, and more are between the new features.

#### Express.js

Express.js is a back-end web application framework for Node.js. It has been called the de facto standard server framework for Node.js. It has been used to create web applications and API. It uses Robust routing. With Express.js it is easy to divide the routes by logical sets and files. It will let you use the passport middleware and EJS template rendering. It supports redirection and rendering to costs.

The Express.js view system supports 14+ template engines.

#### EJS

EJS is a simple templating language that allows you to generate HTML markup with plain JavaScript. There are no strict rules on how to organize things. No reinvention of iteration and control flow. It is just plain JavaScript. Offering fast compilation and rendering of HTML pages, it supports a Sub-template for code reuse. The template uses template tags: <% %> to access sent variables.

Complies with the Express.js view system.

Keep in mind that EJS does not change the DOM element on the fly.

#### Passport.js

Passport.JS [6] is authentication middleware for Node.js. Extremely flexible and modular. A Passport can be unobtrusively dropped into any Express-based web application. A comprehensive set of strategies support authentication using a username and password, Facebook, Twitter, and more.

It uses middleware functions to authenticate and can keep the authentication token on the session or the cookie at the host browser.

### MVC – Model View Controller

A software design pattern. We have divided the project into three logical connected parts:

### Model

The application's dynamic data structure, independent of the user interface, directly manages the application's data, logic, and rules. In some cases, it is the central component of the pattern. As shown in figure 4, the model uses independent parts, the combination of which, creates the ecosystem.

### View

Display the output and the forms data that help the user create reports. Multiple views of the same information are possible.

### Controller

Accepts input and converts it to commands for the model or view.

By using MVC, we can separate each part of the development test it deploys without changing all the code.

Graphical user interface, application, timeline

Description automatically generated

Figure - Model View Controller [14]

# A picture containing text, vector graphics, light Description automatically generatedMGO – Medical Graph Ontology Overview.

MGO Key Features

* Add a Symptoms and Treatments node.
  + Each Symptom connects to treatments using vertices (if it has one). Every Treatment connects to Symptom as a treatment or a side effect.
  + Add a tag to nodes and search by tag.
  + Add data from different sources, we use data node.
* Show each node and its neighbors.
* Analyze data:
  + Run shortest path from 2 nodes.
  + Union of several nodes to look for unseen nodes (Symptoms and Treatments).
  + Run cypher query, direct from the project.
* Add bulk data, a way to enter many data in one action.

Installation

1. Make sure you have Node.js installed.
2. Clone the MGO master branch
   * GH repo clone <https://github.com/itzikorfa/Medical-Graph-Ontology.git> or use GitHub desktop.
3. Connect to MongoDB

* Install MongoDB server if needed. ([link to download](https://www.mongodb.com/try/download/community))
* Create a collection to connect.
* Get the connection string and update the parameter "DATABASE" in config/config.js

1. Connect to Neo4j Graph database

* Install Neo4j server if needed. ([link to download](file:///C:/Volumes/GoogleDrive/My%20Drive/2/MGO/manual/link%20to%20download))
* Create a database.
* Get the connection string and update the parameter in config/config.js:
  + Username
  + Password
  + dbName
  + Port

1. Install dependencies:
   * Run "npm install" at the cloned directory.
2. Start the server
   * You will need to set an environment variable named PORT to run the server on.
   * At the cloned directory, run npm start.
3. Go to the site:
   * Access to your server IP address (if you run it from your local computer, it is 127.0.0.1) http://<server ip>:<server prort>

## MGO Project Functionality

### User Control

When the project is online anyone can view the project content. Look for symptoms and treatments tags, and run availed analysis. But if the user wants to add data or connect different nodes, the user must register and log in to the site.

### Symptom Manager

The user can view all the symptoms as a graph and as a link. The user can choose a symptom and view the status. When the user is logged in, he can add symptoms and manage symptoms by adding data nodes and tag nodes.

A symptom can be added from the treatment view and the user can add side effects to an existing symptom.

### Treatment Manager

The user can view all the treatments as a graph and as a link. The user can choose a treatment and view current tags and symptoms the treatment treat. When the user is logged in, they can add treatment from the symptom view. From the treatments manager in the treatment view, the user can add tags and side effects.

### Tags Manager

At the tag view, the user can look at the list of tags, and by choosing a tag link, they will get a view of all nodes as the requested tag.

### Add bulk data

A view that lets the logged-in user add a JSON data struct that holds symptoms and treatments and the way they are connected. This server receives the JSON to create a transaction.

### Analyze Data

The goal of the project is to analyze the data and find unknown relations.

The project supports the following algorithms:

Union Search – Find and display a graph and node links of a union of nodes (symptoms, treatments, and tags), this view is in a graph and links the nodes.

All Shorts Path – Enter 2 nodes name and their type, the project will then find the connection (if it exists) between the 2 nodes. The project uses the existing Neo4j function of the “all shorts path” algorithm between the 2 nodes. The view contains the node links and the graph.

Cypher Query – Run a cypher query directly from the project. The result is a Cypher Json struct that the system cleans, returning the resulting graph.

## MGO User Guide

### Main Page

When you enter the main page of the site, you will see Figure 1. We can navigate through the site using the navbar (upper horizontal menu), which is the main menu of the project.

You can view a complete graph with connections of the symptoms and treatments graph in the main menu.

On the navbar, you can choose to view only Symptoms, Treatments, or Tags. Alternatively, you can choose to run the analysis screen, which lets you run different types of algorithms.

On the right side, we have a User status button, this is in Login Mode if no user is logged in. If a user is logged in, it changes to a user name and Logout. In the Login Mode, you can also register. When you are registered, login gives you the ability to add new data to the system.

We will throw this ability later in the report.

The last feature that the project allows is to insert bulk information using a JSON data structure. This can be done by pressing the gear icon.

### User Control

I have implemented a minimal Role Base Access Control. Any user can view the data run analytics. However, you will need to register and log in if you want to add data to the system. If we try to access a form when we are not logged in, we will be redirected to a login page that the user can register from this page too.

To register, the user can press the login button on the right side of the navbar and press the register link.

For the RBAC, I used the passport module that has its chapter.

Graphical user interface, application

Description automatically generated

Figure 5- Login page

Graphical user interface, application

Description automatically generated

Figure 6 - Registration page

Graphical user interface, text, application

Description automatically generated

Figure 7- Login page

Graphical user interface, application

Description automatically generated

Figure 8 - Welcome page

In figure 5, we see the popup login that we can access from the navbar. Figure 6 is the registration page. Figure 7 is the login page that will appear after registration or when the user tries to access the page. The user must be logged in to access. Figure 8 is the welcome screen after the user login.

### Symptoms

When we select Symptom's link at the navbar, we will see a window showing all the symptoms entered into the system (Figure 2).

We can choose a symptom and view its current information and graph (Figure 3).

We can also view the treatments, the tags, and the data. (Figure 4, Figure 5).

#### Adding a New Symptom

To add a new symptom:

* You must be logged in to the system.
  + If you are not logged in, the system will take you to the login form.
* Click on Symptoms at the navbar.
* Click add new Symptom link
* Fill in all the required information.
  + To add more than one Tag, place a comma between the tags.
* Click on add Symptom.

#### Symptoms Data

As part of the research, we found different sources for every Symptom. We want to be able to keep the different data. We created a new Node type named Data, and every Symptom can have 0…n data nodes.

#### Add data to Symptom

* Click on symptoms,
* Find and select the Symptom
* Click on data 🡪 add data.

Figure - Symptoms view

Graphical user interface, text, application, chat or text message

Description automatically generated

Chart

Description automatically generated with medium confidence

Figure 10- Symptom view

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 11-Symptom information

Graphical user interface, text, application, email

Description automatically generated

Figure 12- Add symptom form

Figure 9 shows the Symptoms view that shows all the symptoms by clicking the symptom link symptom information view (figure 10, and figure 11). Figure 12 shows the added symptom form.

### Treatments

When we select the treatment's link at the navbar, we will see a window that shows all the treatments that are entered into the system (figure 13).

We can choose a treatment and view its current information and graph (figure 16).

We can view the symptoms and the tags, and the side effects.

Treatments have no extra data (figure 17).

Add existing treatment – The same treatment is used for a couple of symptoms. After creating a new treatment, we can add it to existing symptoms (figure 14).

Treatments Side Effects- are another symptom that already exists.

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 13- All treatmentsGraphical user interface, application

Description automatically generatedGraphical user interface, application

Description automatically generated

Figure - Connect treatment to existing symptom

Figure - Add treatment

Graphical user interface, text, website

Description automatically generatedDiagram

Description automatically generated

Figure - Treatment information

Figure 17-Treatment graph

### Tags

Tags are a way to categorize symptoms and treatments. Using the tags mechanism, the user can get all symptoms/treatments tagged with the same tag.

Figure 18 shows tag view and the node link.

#### Tag Implementation

To implement the tag mechanism, I used the graph database with a new node type, name tag and a relation tag. All added tags are transformed to uppercase.

#### Adding a New Tag

Tags are not independent objects and can only exist on an existing object.

The only way to add a tag is when adding a symptom or treatment.

When we select the Tag's link at the navbar, we will see a window showing all the tags in the system.

We can choose a tag and view its current information and graph.

A picture containing application

Description automatically generated

Figure 18- Tag view

### Bulk Actions

Let's say we want to add many objects. We can enter the MGO project login and add data manually step by step. However, I have added new support to add a JSON object that holds a lot of struct information. The data in the bulk data will be entered as a transaction, and in case of an error, a rollback will occur. All the new nodes must be new information.

#### Bulk JSON Data Structure

##### Main Keys

* Create – to create new nodes.
* Data – add data nodes to a symptoms
* Connect – to connect new nodes.

Create – a list of nodes that have the following keys:

* Type – a type of the node to create.
* Name – the name of the node.
* Common\_name – a short name for the node.
* Desc – description of the node.
* Tags – a string of tags with a comma between the tags.

Data – a list of data nodes to be added to the new symptoms. The keys that the data must have are:

* Symptom – the name of the new Symptom.
* Header – the header of the data node.
* Data – data to be added.

Connect – A list of edges between symptom and treatment, the required edge keys are:

* Symptom – the name of the symptom node.
* Treatment – the name of the treatment node.
* Connect\_type – the type of connection between the two nodes, it can be:

treatment or Side\_effect.

#### How to Add Bulk Data

* Prepare the data in a text editor.
* It recommends validating that the JSON object is valid.
* Click on the gear icon on the left side of the navbar.
* Click on JSON input.
* Paste the JSON object to the JSON data text input.
* Click on Add Data button.

## Analyze Data

The main goal is to analyze the data we have collected. I have created two main methods to analyze data, UNION Search and All Shorted paths. I added one more tool to run Cypher queries directly from the site to get the JSON results.

### Union Search

Goal:

To identify treatments and symptoms that are related to a set of symptoms and treatments. If a patient has some symptoms that they cannot find a reason for, he can choose a list of symptoms and have some treatments and tags. The output is all the nodes that are related.

#### How to Run Union Search

* Select analyze from the main navbar.
* Select the Union Search link (figure 19).
* Start adding symptoms treatments and tags.
* Press the Search button.

Graphical user interface, text, application

Description automatically generated

Figure - Union search input

#### The Output

Graphical user interface, text, application, chat or text message

Description automatically generatedYou can view the set of symptoms and treatments that the algorithm found. Click on each of them to find more information (figure 20 and figure 21).

Figure - Union search output

A picture containing chart

Description automatically generated

Figure -Union search output

### All Shorts Path

#### Goal:

Find the relation between 2 nodes and display all the nodes in the path.

In this analysis method, we need to specify the node type that we need to query, and then select the source and destination node.

#### How to Run All Shorts Path:

* Select analyze from the main navbar.
* Select the All Shorts Path link
* Select Source Node type and destination source type.
* Select nodes name.
* Press the Search button.

#### The Output

Figure 23 shows the shorts path output view. The set of symptoms and treatments that the algorithm found. Click on each of them to find more information.

Chart

Description automatically generated

Figure - Shorts path output

### Cypher Queries

##### Goal:

Graphical user interface, application

Description automatically generatedRun a Cypher query directly from the project. This way, the user does not need to install the Neo4j Management and connect to the Neo4j Server.

Figure - Cypher input window

##### How to Cypher Query

* Select analyze from the main navbar.
* Select the Cypher queries card link (figure 23).
* Write the Cypher query and the parameters (if you need parameters).
* Press the Search button.

A picture containing text

Description automatically generatedFigure 24 shows the JSON result window of a query.

Figure - Cypher query result

##### The Output

It is a JSON output (in collapse mode), that can be copied and analyzed.

# The Code

## The Directory Structure:

The project design in the MVC module. The main directories are:

* Controller – hold the logical algorithms to create the data to generate the output.
* Models – the dynamic data structure class that holds the primary data structure.
* Views – the EJB HTML templates.
* Routes – hold the routes files. These files create the subdomains.
* Config – hold the configuration files to connect to the Neo4j and MongoDB.
* DB – the Neo4j middleware connection.
* Public – the static HTML,CSS, JS.

[The code snippet in the appendix](#_The_main_files)

# **Summary and Future Development**

The project gives you a perfect base for future development, It provides important functionality like viewing/adding symptoms, treatments and data. It can be extended to many medical categories and domains

The project enables a perfect base for future development, adds a need for queries and analyzes methods. I want to continue developing the project and maximize its abilities.

Next steps in development:

* API ability to run any algorithm that MGO supports.
* Support more algorithms to run on the Graph.
* Connect to available medical data centers.
* Share results.

## **Appendix A – The software main modules**

## The Main Files

### ./package.JSON

Package.json [11] This file holds various metadata relevant to the project. This file is used to give information to npm that allows it to identify the project and handle the project's dependencies. It contains other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data.

When we want to install all dependencies of a node project, all we need to do is to run the command:

npm, install – this command would look for the package.json and get from npm library all the needed dependencies recursively.

#### Script Parameter

In the run mode that the server starts, every mode will run in different switches and scripts.

### ./app.js

The file in the root directory of the project, is the leading runner that stats the server. It constructs into four parts:

1. Set global application parameters – when the server starts, it needs the template's location, start logging, define the MongoDB connection, and more.
2. Define application middleware – a middleware is a function that will run between 2 steps of the application. The application uses passport middleware for user access and notification to the user.
3. Register application routes – router routes registration, connect a route to a route file.
4. Start the server – get the port address and start the server and the central tread.

#### ./.env file - DotEnv module

Some parameters need to be defined from the project's environment variables (server port and database URI). When using a development environment, we cannot set all environment parameters that can collapse between different projects. We use the dotenv module that reads the environment parameters from the .env file.

### ./config/passport.js

Set the login strategy that the project support. We can add any Passport module support method (using Google, Twitter, etc.)

Our project supports the Local Strategy that uses the user name and password hash to authenticate the user.

### ./config/auth

The middleware functions user requests permissions to the resources.

Functions:

ensureAuthenticated – returns a middleware that will check if the user is logged. If not, it will redirect the request to the login page with the correct message.

forwardAuthenticated – if the user is not authenticated, it will continue to log in; else (the user has succeeded in logging in), redirecting the response to the dashboard view.

### ./db/neo\_connection.js

The main file to run the Cypher query on a Neo4j server. The function will return the total result to be analyzed.

The function:

run – receive the query to run and the parameters that the function needs to use.

### ./models/User.js

const UserSchema = new mongoose.Schema({…})

The User mongoose schema that connects to MongoDB.

In the schema, we define the user parameters and validation. If we want to add more parameters to the user DB, we need to add the parameters to the schema.

### ./models/Queries.js

a JSON file that holds all the queries that the project uses.

As we used a generic way to run symptoms and treatment queries, a "\_\_" text is used instead of defining the node type. Using string manipulation, we replace the node type.

### ./models/BaseNode.js

#### class BaseNode {..}

It is the model class for the symptoms and treatments, and it helps get an existing node to create a node and more.

#### constructor(node\_type){..}

The class constructor gets the node type parameter that the instance will be. It can be a symptom or treatment.

#### init(name, common\_name= "", description="", id=null){..}

The init function will initialize the node parameters. For the object to init, we must provide a name (it is unique to a type); all other parameters are not mandatories. Only after we initialize the object, can we run the class methods.

#### add\_node(){..}

A function that adds a new node to the graph DB if a node with the same name exists. The function will return to the home directory with the correct message.

The flow of the function is:

* Check if the node already exists.
* Create Node.
* Create Tags if they do not exist and connect them to the node.

#### add\_data(header, data){..}

Create a data node and connect it to the symptom node.

#### add\_tag(tag\_name){..}

Add a tag to an existing node. Uses the TAG class.

#### connect\_nodes(target){..}

Connect treatment and symptom, and the edge type is set by how to connect. If treatment connects to the symptom, the edge type is a symptom. If the node type is treatment, the edge type is a treatment.

#### static get\_by\_id(name, node\_type){..}

Get a BaseNode object by name and type, it will return a JSON object that includes all connected nodes including TAG’s,data, and connected Basenodes.

#### static get\_all\_nodes\_name(node\_type){..}

Return a list of names by node type.

### class Data{..}

The data class represents the data node that connects to the symptom.

#### init(header, data, basenode, id = null) {..}

Init the object with the data header and data information. Moreover, the base node to connect.

#### create() {..}

Creates the data node.

#### connect() {..}

Connect to data node to a basenode.

### ./models/Tag.js

The Tag class is responsible for all tags operation.

#### init(name){..}

The init function will initialize the node name parameter.

#### add\_tag(bnode){..}

The method adds a new Tag node if it does not exist and connects it to the existing node.

#### static add\_tags(arr\_tag, bnode) {..}

The function receives an array of tags and adds them to the Neo4j DB.

In this method, I used the promise. All methods that will run all promises function and will reject if only one on the set rejects.

#### static get\_tag(name){..}

The function that gets all connected nodes to a tag.

### ./routes directory

The routes directory holds all endpoints of the project. The structure of each route file is:

* At the head of the file, we import the needed function from the controller.
* Creating a route object.
* Add to the route endpoints with the endpoint name and function to be used.
* Adding needed middleware like ensure Authenticated (check if the user is authenticated before running the function).

### ./controller directory

The symptom controller receives the requests, processes the information, and sends the response. Most exported functions get the requested parameters, get the information, and return it to the EJB template. Here are some unique functions:

### ./controller/bulk.js

The script that holds all the bulk action algorithms. The data in the request must be a JSON data structure, all the node and connection variables must be unique in the transaction. To handle these issues, I have created the current flow of tasks:

1. Check JSON structure.
2. Validate that all fields exist.
3. Add nodes queries and save variables.
4. Add tags queries and save variables.
5. Connect nodes and tags
6. Create data if needed and save variables.
7. Connect nodes
8. Run query.

All steps update the temp file and, in the end, read the queries and run to save the transaction.

### ./controller/analyze.js

The main file that holds all the analysis algorithms and requests arrive in the analyze route and passes the requests to the analysis.

#### Union Algorithm

Our goal, as mentioned is to find all the nodes that the user entered.

##### function union\_add\_queries(query\_type, data\_arr, data){..}

Receive a list of nodes and return a query that will display all nodes and connections to the nodes with parameters.

##### exports.union = async (req,res)=>{..}

The request holds three arrays of symptoms, treatments, and tags. Every set array sends to union\_add\_queries function, the function return dictionary that holds the quey and parameter query.

The algorithm joins the queries, and the parameter runs the query on the Neo4j DB and sends the output to the template.

The query sends to the DB and a query to the EJB template to display the graph.

#### Shortest Path Analyze

Our goal is to find a connection between 2 nodes and check for an unknown cause.

##### shortest\_path\_analyze(data){..}

The function receives the requests and gets the source node and type and destination node and type. The function generates the query, runs it on the Neo4j, and analyzes the results.

The function generates an entire query to display the graphs.

#### Cypher Query from the Portal

If you need a more generic way to analyze the data, I gave the user a way to run a query directly from the project.

##### exports.cyperfreequery = async (req,res)

Receive a query and a JSON string, validate the JSON, and the query will not harm the data and run the query.

Next, results need to be cleared of unnecessary information and only return the nodes and relations. I ran a function that removes this information and returns the JSON. Moreover, build a query to display the graph data.

# **Appendix B – Review of main concepts**

## Ontology

Ontology in information science is a way to represent a domain using the definition of the categories, properties, and relations. [7, 6]

Ontology is used in different academic disciplines or fields. The way the data organize helps them understand and limit complexity and organize data into information and knowledge.

In computer science, information science, and systems engineering, ontology engineering is a field that studies the methods and methodologies for building ontologies

The way ontology organize data helps artificial intelligence decision making,

the ontology may be specified formally:

Where:

Figure 25 – [8]

Display an ontology graph.

Diagram

Description automatically generated

Figure 25- ontology

### Ontology Usage

There are many ways to use an ontology, and in time there were different query languages that were developed to help to get data from the ontology. I will elaborate on two significant programs/languages.

### SPARQL

SPARQL (SPARQL - Wikipedia, n.d.) is a semantic query language for databases to retrieve and manipulate data stored in RDF (Resource Description Framework) format. It was made a standard by the RDF Data Access Working Group (DAWG) of the World Wide Web Consortium and is recognized as one of the critical technologies of the semantic web.

RDF was defined as a W3C recommendation in 1999

SPARQL allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns.

SPARQL is linked to RDF when it comes to data types, operators, and functions so that SPARQL can do, e.g. string and numerical operations.

For SPARQL to work, we need the data to be defined of the subject and a statement about it. For example:

<http://somewhere/MattJones/>

vCard:FN "Matt Jones" ;

vCard:N [ vCard:Family

"Jones" ;

vCard:Given

"Matthew"

].

The subject of the vCard is Matt Jones, that has the family name Jones and a given name Matthew.

SPARQL can access the data and provides a full set of analytic query operations such as JOIN, SORT, AGGREGATE. This can be performed for data whose schema is intrinsically part of the data rather than requiring a separate schema definition.

For example, we want to find a card named John Smith:

SELECT ?x

WHERE { ?x <http://www.w3.org/2001/vcard-rdf/3.0#FN> "John Smith" }

Most SPARQL users use Stardog studio to run queries on the existing data.

figure 2 – shows a Stardog IDE.

### Resource Description Framework

RDF (Resource Description Framework, n.d.) - Classical conceptual modeling approaches to help access the semantic data world. In RDF, we change any data to statements about resources. The subject denotes the resource, denotes traits or aspects of the resource, and expresses a relationship between the subject and the object. As a definition, ontology is data modeling using categories, properties, and relations.

### Protégé

Graphical user interface, application

Description automatically generatedProtégé [9] - Program for an ontology development environment, supported and been used by a strong community of academic, government, and corporate users. Protégé is used to analyze ontology and integrate the output with rule systems or other problem solvers to construct a wide range of intelligent systems.

Figure - Protégé main window

Figure 26 shows the main window of a project, on the right side we can see the ontology metric.

Protégé fully supports the latest OWL 2 Web Ontology Language and RDF specifications from the World Wide Web Consortium.

Protégé is written in JAVA and uses XML data files.

Deductive classifiers are included in one of the first ontology editors with a graphical user interface to evaluate coherent models and infer new information based on data analysis.

Protégé lets you access local OWA files or get data from a remote site, and we can union two ontologies. It supports class hierarchy and redefines a relation between the class. The product can be viewed as a tree or as a graph relation.

For every class, Protégé creates a dynamic form to input data.

Protégé can get data from SPARQL queries and display the data.

#### Create a New Ontology

To create a new ontology using Protégé, you need to:

* Define the class and the class hierarchy.
* Create data properties.

Data properties - Information that all class object holds.

* Create object properties that create the composition relationship between objects.

Timeline

Description automatically generatedThe more data we have, the more accurate the ontology is.

Figure - Protégé class hierarchy

### Class expressions

Suppose we need to describe individuals with common characteristics. In that case, we use Class expressions, which use 2 class types and expressions that will return the information we looked for. The expression can be complex and nested.

To run Class expressions, go into the DL query section. In the class, the expression text area enters the needed information. Figure 27 shows where we can view the class hierarchy, here we can add a new class.

Figure 28 shows the class properties that are set by the user or by the Protégé analyst. Figure 29 shows that we can see one of the views that show the ontology as a graph.

Graphical user interface, text, application, email

Description automatically generatedDiagram

Description automatically generated

Figure - Protégé class properties

Figure - Protégé Graph view

## Graph Theory

Graph theory mathematical structures are used to model pairwise relations between objects. It is made of vertices (Called nodes, points, objects) and edges (also called link, line, or relations). The vertices must have a name to identify them and different types of properties.

The edges must have a type that identifies the type of relation and different types of properties.

When there is an edge of type {} between 2 nodes, there is a relation {𝛼} between the 2 nodes. If there is no edge of this type, it means that the nodes do not hold this type of relation.

The formal way to define a graph:

G – graph

V – set of vertices

E – set of edges

### Graph Categories:

Directed and Undirected Graph – A directed graph means that if there is an edge from node A to node B, that does not mean that we have the same relation between node B to node A. If we want a relation between node B to node A, there must be a different edge. The arrow represents the direction.

Weighted Graph – It is a graph that has a weight on the edges. A weight is a number that represents the cost of using the edge if we use the current route.

We can represent almost every ecosystem/situation.

The edges can be a direct graph or an undirect graph.

The graph can weigh the edges that describe the nature of the relationship between two nodes.

Graph Algorithms – A way to solve different types of problems using the graph data structure. By describing the ecosystem as a graph, we can ask hard questions that the way to solve this issue is very hard. Using graph algorithms makes the answer a lot quicker, and the answer can be clearer to understand.

## Database

A database is an organized collection of structured information, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). Together, the data and the DBMS, along with the associated applications, are referred to as a database system, often shortened to just a database.

### Query Language (QL)

A computer programming language that requests and retrieves data from the database management system sending queries. Usually, user-entered structured and formal programming command-based queries to find and extract data from host databases.

### Database Types

There are several different types of databases. The main categories are:

Relational Database – A database that stores and provides access to data points related to one another. All the rows in a table have the same fields. You can create constraints when entering data or a relation. The data and the relationships are stored in logical tables.

The relational databases use Structured Query Language (SQL).

The leading vendors are – MySQL, MS-SQL, SQLite, Oracle DB.

Document Database - A database that holds documents (usually JSON objects). It uses the primary dictionary definition {key, value}, not all records must have the same fields. The main advantage of Document Database is the fast insertion.

The Document Database uses Document Query Language, which can handle all CRUD operations.

Top vendors are MongoDB, AWS DynamoDB, IBM Cloudant.

## Neo4j Graph Database

Graph Database – [information in the article](#_Graph_database).

### Neo4j Storage

[10]Use Neo4j database files on storage, using Native Graph-Storage Model which uses pointers.

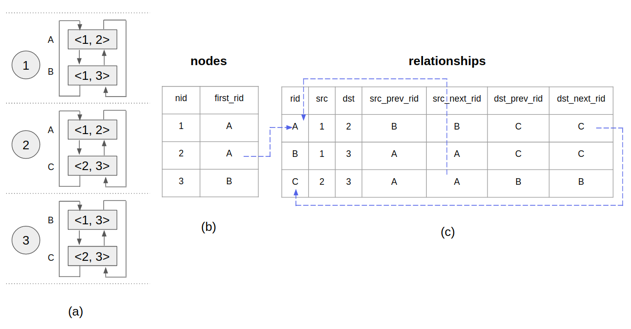
As shown in figure 30, data stored on a disk are all linked lists of fixed-size records. Properties are stored as a linked list of property records, each holding a key and value and pointing to the next property. That way, we can implement the Graph as a Linklist.

Figure - Neo4j storage module [10]

### Neo4j Licenses Type

Neo4j is an open-source project whose source and copyright are owned and maintained by Neo4j, Inc.

There are two main versions:

Neo4j Community Edition - Fully-featured, best-in-class graph database that uses the GPL v3 license. Some features are locked in the community version (like User Role management).

Neo4j Enterprise Edition - designed for commercial deployments where scale and availability are essential.

* Includes all features from the community edition.
* Unlimited horizontal scaling with replication and shading.
* Fine-grained access controls.
* High availability and advanced manageability.
* Available as an operational cluster or stand-alone data science environment.

The enterprise edition can come in 3 different licenses:

* Neo4j Commercial License
* Neo4j Developer License
* Neo4j Evaluation License

# Figures Table

[Figure 1 - Project Components Flow 6](#_Toc95690281)

[Figure 2- MGO Main Window 8](#_Toc95690282)

[Figure 3-Neo4j Desktop Window 12](#_Toc95690283)

[Figure 4 - Model View Controller [14] 16](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690284)

[Figure 5- Login page 20](#_Toc95690285)

[Figure 6 -Registration page 20](#_Toc95690286)

[Figure 7- Login page 21](#_Toc95690287)

[Figure 8 - Welcome page 21](#_Toc95690288)

[Figure 9 - Symptoms view 22](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690289)

[Figure 10- Symptom view 22](#_Toc95690290)

[Figure 11-Symptom information 23](#_Toc95690291)

[Figure 12- Add symptom form 23](#_Toc95690292)

[Figure 13- All treatments 24](#_Toc95690293)

[Figure 14 - Connect treatment to existing symptom 24](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690294)

[Figure 15- Add treatment 24](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690295)

[Figure 16- Treatment information 25](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690296)

[Figure 17-Treatment graph 25](#_Toc95690297)

[Figure 18- Tag view 26](#_Toc95690298)

[Figure 19 - Union search input 28](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690299)

[Figure 20- Union search output 28](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690300)

[Figure 21-Union search output 28](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690301)

[Figure 22- Shorts path output 29](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690302)

[Figure 23- Cypher input window 30](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690303)

[Figure 24- Cypher query result 30](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690304)

[Figure 25- ontology 33](#_Toc95690305)

[Figure 26 - Protégé main window 35](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690306)

[Figure 27- Protégé class hierarchy 36](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690307)

[Figure 28- Protégé class properties 37](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690308)

[Figure 29- Protégé Graph view 37](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690309)

[Figure 30 - Neo4j storage module [10] 40](file://Volumes/GoogleDrive/My%20Drive/2/MGO/הגשה/Orfanian_Yizhak_Advanced_project_031860984_ver2.docx#_Toc95690310)

# Bibliography

|  |  |
| --- | --- |
| [1] | "wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/MongoDB. |
| [2] | "Storage," [Online]. Available: https://docs.mongodb.com/manual/storage/. |
| [3] | A. V. N. W. Jonas Partner, Neo4j in Action, Manning Publications. |
| [4] | "Wikipedia - Node JS," [Online]. Available: https://en.wikipedia.org/wiki/Node.js. |
| [5] | "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Npm\_(software). |
| [6] | "Passport," [Online]. Available: http://www.passportjs.org/docs/downloads/html/. |
| [7] | "Ontology (information science)," 10 08 2021. [Online]. Available: https://en.wikipedia.org/wiki/Ontology\_(information\_science)#References. |
| [8] | "Examples of Ontology Model Usage in Engineering Fields," 11 08 2017. [Online]. Available: https://www.intechopen.com/chapters/59449. |
| [9] | "Protégé," [Online]. Available: https://protege.stanford.edu/. |
| [10] | "Neo4j storage model," [Online]. Available: https://neo4j.com/developer/kb/understanding-data-on-disk/. |
| [11] | "package.json," [Online]. Available: https://nodejs.org/en/knowledge/getting-started/npm/what-is-the-file-package-json/. |
| [12] | "SPARQL - wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/SPARQL. |
| [13] | "Resource Description Framework," [Online]. Available: https://en.wikipedia.org/wiki/Resource\_Description\_Framework. |