Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Graph Database Fundamental Services

Tomáš Roun

Supervisor: RNDr. Marko Genyk-Berezovskyj
Field of study: Open Informatics
Subfield: Computer and Informatic Science
May 2018

Acknowledgements

I would like to thank my advisor RNDr. Marko Genyk-Berezovskyj for his guidance and advice. I would also like to thank Sergej Kurbanov and Herbert Ullrich for their help and contributions to the project. Special thanks go to my family for their never-ending support.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ....

....
signature

Abstract

The goal of this thesis is to provide an easy-to-use web service offering a database of undirected graphs that can be searched based on the graph properties. In addition, it should also allow to compute properties of user-supplied graphs with the help graph libraries and generate graph images. Last but not least, we implement a system that allows bulk adding of new graphs to the database and computing their properties.

Keywords: Node.js, Wolfram Mathematica, SageMath, NetworkX, Graphviz, PosgreSQL, Graph theory

Supervisor: RNDr. Marko Genyk-Berezovskyj
Faculty of Electrical Engineering, Department of Cybernetics, Na Zderaze 269/4, 120 00 Prague 2

Abstrakt

Cílem této práce je vyvinout webovou službu nabízející databázi neorientovaných grafů, kterou bude možno efektivně prohledávat na základě vlastností grafů. Tato služba zároveň umožní vypočítávat grafové vlastnosti pro grafy zadané uživatelem s pomocí grafových knihoven a zobrazovat obrázky grafů. V neposlední řadě je také cílem navrhnout systém na hromadné přidávání grafů do databáze a výpočet jejich vlastností.

Klíčová slova: Node.js, Wolfram Mathematica, SageMath, NetworkX, Graphviz, PosgreSQL, Teorie grafů

Překlad názvu: Základní služby grafové databáze

Contents

Figures

Tables

Chapter 1
Introduction

This project is a collaboration between several students. My task is to develop the backbone of the application which provides the graph database. This entails, in no particular order, writing a HTTP server that is accessible from the internet and designing an API to communicate with it, maintaining a database containing the graphs themselves, integrating graph libraries that are be used to compute graph properties and deploying the application to a server. Secondary goals include writing a documentation and tests. Sergej Kurbanov's task is to create a user interface as described in his thesis [21], while Herbert Ullrich investigates the problem of isomorphism as described in his thesis[22].

1.1 Motivation

There are currently only a handful of online services that offer a searchable database of undirected graphs and all of them have limitations which make their use case somehow limited. Services like House of Graphs[8] or Encyclopedia of Graphs[2] only collect certain classes of graphs or graphs that have been deemed interesting in some way. Our goal is to provide a database of all graphs up to a certain number of vertices regardless of their intrinsic value. We feel that having a complete list of graphs up to a certain number of vertices can prove very valuable to researches. At the same time, certain classes of graphs can be optionally included in the database as well.

While there exist searchable graph databases, we are not aware of a general-purpose web service that would allow users to compute properties of user-supplied graphs. Because of the sheer number of non-isomorphic graphs, even on just 11 and 12 vertices (1018997864 and 165091172592), it is impossible to categorize even just a small percentage of graphs and arguably, a lot of interesting graphs have much more vertices. This is a big limitation for most of the graph databases, where users can only search through the very limited list of categorized graphs. The idea of this project is to accept the fact that we will only ever able offer just a tiny portion of all graphs up to a certain size and instead, let users upload their graphs and have our service compute graph properties for them.

1.2 Thesis Structure

The thesis structure is described below:

- Chapter 2 presents an overview of the intended features of the application.
- Chapter 3 describes the software tools and technologies we decided to use to develop this project as well as a light introduction into the way the tools and technologies interact with each other.
- Chapter 4 explains the inner workings of the **graph6** format that the project relies on heavily.
- Chapter 5 presents the API design of the web server application. The chapter describes the most important API endpoints as well as the specification, including parameters, status codes and overall behavior.
- Chapter 6 describes the implementation, the application flow and the way the web server, the database and the graph libraries work together. We also analyze the speed of the graph libraries on selected graph properties.
- Chapter 7 describes the notion of software testing and code coverage and the way it helps us ensure the correctness of the application.

Chapter 2

Features

In this chapter we describe the basic features of the project the way they are implemented.

2.1 Searchable database

We base the graph service on a model which House of Graphs[8] uses. House of Graphs allows users to query the database based on complex conditions relating to the graph properties. For example, one can search for graphs with a certain number of vertices in conjunction with a certain number of edges. Our goal is to design an API that is expressive enough to encode such complex queries, but which can also be safely translated into SQL. The database should be able store large number of graphs and allow fast lookups.

In chapter 5, we describe the API that is used to communicate with the database.

In chapter 6, we describe the database schema and the overall design of the database.

2.2 Online Computation

As mentioned in chapter 1, there do not exist any web-based services that allow users to upload their own graphs and have the service compute the graph properties. However, there are many graph theory libraries that can be used offline to achieve the same result. Our goal is to select a few of these libraries and integrate them with the web server so that they can be used to compute properties online. We selected three graph libraries for this purpose.

- Wolfram Mathematica – Mathematica is a technical computing system encompassing a large number of areas of mathematics and numerical computation one of which is a support for many graph theory computations. Mathematica is a commercial closed-source system and uses a proprietary programming language.
- SageMath – SageMath is a computer algebra system covering many aspects of mathematics including algebra, combinatorics, numerical

analysis and graph theory. It is fully open-source and written in a combination of Python, C and C++.

- NetworkX – NetworkX is a Python library created for studying graphs and networks featuring many graph algorithms and network analysis tools. Just like SageMath, it is fully open-source and free to use.

The number of graph properties that can be computed on graphs is too large to cover completely. We focus mainly on the most common or interesting graph properties in this project. A lot of the graph properties are difficult to compute, many of them having exponential complexity, so it is important to realize for larger graphs the list of available graph properties will be necessarily reduced, not only due to the aforementioned theoretical complexity but also due to inefficient software implementations.

In chapter 5, we describe the API that is used to communicate with these libraries over the HTTP protocol.

In chapter 6, we describe how these libraries are implemented in the project and also present a speed comparison of the graph libraries.

2.3 Graph Visualization

We want users to be able to view the images of the graphs we store in the database as well as the graphs they input via the online computation service.

Since drawing graphs is generally a difficult problem (the crossing number problem which determines the lowest number of edge crossings of a plane drawing of a graph is NP-hard), we will use a proven tool called Graphviz[7]. Graphviz is a visualization software that can produce graph images in many formats relatively quickly. It includes several layout programs, also called engines, each of them based on a different computational model:

- dot – This is the default engine recommended for directed graphs.
- neato – neato is the default engine recommended for undirected graphs.
- fdp – fdp, same as neato, is supposed to be used for undirected graphs. It works by reducing forces in a spring model.
- twopi – Creates radial layouts where vertices are placed on concentric circles.
- circo – Creates circular layouts, suitable for cyclic structures.

In chapter 5, we present the API that interfaces with Graphviz and the static images of well-known graphs.

In chapter 6, we describe the implementation and compare the speeds of the Graphviz engines.

2.4 Automated Database Maintenance

As a part of this project we develop a command line utility, whose purpose is to automate tasks associated with database maintenance. First, the utility should be able to display the current status of the database, showing useful information about the graphs and the graph properties. Second, the utility should be able to import new graphs, while ensuring that isomorphic graphs are filtered out. This part relies on a tool written by Herbert Ullrich as a part of his thesis[22]. Lastly, the utility should be able to compute missing values in the database using one of the graph theory libraries.

In chapter 6, we describe the implementation and the typical usage of this utility.

Chapter 3

Technologies & Tools

3.1 Web Server

Our service is web-based, so our goal is to find a suitable technology stack that can be easily integrated with both a database and external graph libraries, while not adding too much complexity to the project. There are many frameworks in many programming languages that are capable of such task. We decided to use Nginx[11] and Node.js[12] coupled with Express[3]. We describe Nginx in section 2, Node.js in section 4 and Express in section 5 of this chapter. The main reason for choosing Node.js was that We have already been using it for some time. Another important reason for choosing Node.js is that the language it uses is Javascript, which will allow us to implement the backend and frontend in the same language and thus we will be able to share code and libraries among all the parts of the project.

3.2 Nginx

Nginx is a high-performance Web server and reverse proxy with built-in load-balancing capabilities. Nginx was released in 2004 after being developed for two years by Igor Sysoev. The goal was to create a software solution capable of handling 10000 requests per second while having minimal system requirements and flexible configuration. Nginx is configured as a reverse proxy for the Node.js server. The reason to add Nginx and not just simply use Node.js is that Nginx allows higher configurability in terms of the HTTP protocol and can be used to balance load between multiple Node.js instances. If there is ever a need to migrate the underlying Node.js application, Nginx can simply be configured to point to the new IP address.

3.3 Javascript

Javascript[1] is a scripting programming language created by Brendan Eich at Netscape in 1995. It was developed as a direct competitor to Sun's Java and both its name and design resemble it. Over the years, Javascript has become the de facto language of the Web and together with HTML and CSS plays an integral part in all modern websites. Javascript is supported in all major browsers, though different vendors support different version of the language. Javascript is a dynamic language, meaning a variable type is determined by its value at runtime. This behavior coupled with the fact that Javascript is a scripting language makes it a very powerful language that allows programmers to achieve a lot in a short amount of time. Javascript, while being an Object Oriented Language is also the only mainstream language that utilizes a so-called prototypical inheritance. The difference from the classical inheritance is that an object does not inherit from a class, but rather from another object that is its "prototype". Another feature that makes Javascript a very useful and well-rounded language is that, apart from being OOP, it is also functional. Functions in Javascript are first-class objects. They can be passed to another function as arguments, as well as returned from functions as values. Javascript makes writing functional code very thanks to the possibility of inline functions as well. There have been several major revisions throughout the history that brought new features to the languages. The most recent and notable ones are ES6 and ES7.

3.3.1 ES6

ES6 also known as ES2015 is a new revision of the Javascript language that adds many new features to the language. The were several important additions to the language in this version:

let & const

Prior to this change, the most common way to declare a variable was to use "var" keyword. The problem with "var" is that it behaves in non-intuitive ways – namely, var declarations are hoisted and function-scoped instead of block-scoped. "let" is supposed to fix these issues and completely replace "var", while const is a new addition that declares constants.

Arrow Functions

Javascript is a functional language by design and it is not unusual to see anonymous functions passed as arguments. The major downside of anonymous function is that their declaration is relatively long. Arrow functions allow us to write less code by introducing a new shorter way to declare anonymous functions. This makes callback functions much more concise and easier to read.

Promises

Before the introduction of promises, asynchronous programming oftentimes lead to a pattern known as "callback hell", which is essentially several levels of nested callback functions:

```
asyncFnA(resultA => {
  asyncFnB(resultA, resultB => {
    asyncFnC(resultB, resultC => {
      // do something with resultC
    })
  })
})
```

Listing 3.1: Nested Callback Functions.

Promises help us break up this pattern into a more manageable linear structure. With the help of promise chaining, the following example is functionally equivalent:

```
asyncFnA()
  .then(resultA => asyncFnB(resultA)
  .then(resultB => asyncFnC(resultB)
  .then(resultC => { /* do something with resultC */ })
```

Listing 3.2: Promise Chaining Example.

3.3.2 ES7

ES7 has introduced the concept of asynchronous functions to the language. Previously, dealing with asynchronous code required either callbacks or Promises or the combination of both. ES7 introduced a third way – the **async** and **await** keywords. By marking a function *async* we can synchronously execute asynchronous code inside of the function by using the *await* keyword. This a very useful addition to the language that has simplified working with non-blocking code. The examples 3.1 and label=lst:prom can be rewritten to with the **async** and **await** keywords:

```
async function run() {
  const resultA = await asyncFnA()
  const resultB = await asyncFnB(resultA)
  const resultC = await asyncFnC(resultB)

  return resultC
}

run()
  .then(resultC => { /* do something with resultC */ })
```

Listing 3.3: async/await Example.

3.4 Node.js

Node.js[12] is a Javascript runtime built on Chrome's V8 engine. It uses an event-driven non-blocking I/O that makes writing code with Node.js very different from other mainstream language platforms. Node.js was originally created in 2009 by Ryan Dahl, with the intention to allow simple creation of Web servers and other network tools. In 2010, a package manager called npm was released, that made it easier to share and manage Node.js libraries.

3.4.1 Asynchronous Code Execution

Node.js does not support a true concurrency like many other languages/platforms do. It is instead single-threaded by design, similar to Python. To get around this limitation, Node.js introduces the concept of asynchronous functions and callbacks. By the single-threaded nature of Node, no two pieces of Javascript code can run at the same time, however some functions, especially I/O heavy ones, can be what is called asynchronous, meaning the I/O part is executed in the background while other Javascript code is running. In a true single-threaded synchronous code a function that opens a file and reads it would block the execution of the rest of the program even though most of the time is spent waiting for the OS and accessing the disk. In Node.js, this is not an issue because while the function is waiting for the disk, other code can be executed. When the function is done reading the file, the function executes a callback that was passed to it in the form of function argument. A callback function is a function that is passed to an asynchronous function and is invoked with the results after the asynchronous function is completed.

3.5 Express

There is a large amount of web frameworks available for Node.js, the most notable being: Express, hapi, Sails.js and koa.js. Most of them achieve the same goal in a slightly different way so it is very difficult to choose between them. In the end, we decided to use Express since just like with Node.js itself, we are already familiar with it and Express has a large active community, tools and libraries behind it. At the same time, with Express, it is very easy to get a simple web server up and running very quickly with minimal amount of code. The following example creates a simple web server that responds to all incoming HTTP GET requests with "Hello, world!".

```
const express = require("express").
const app = express()

app.get("/", (req, res) => {
  res.send("Hello, world!")
})
app.listen(3000)
```

Listing 3.4: Express Server Example.

Besides frameworks, there is a possibility to write an HTTP server in pure Node, after all Node.js was created specifically for this purpose. The reason we chose not to write our own HTTP server, is that the HTTP libraries that Node.js provides are fairly low-level which makes any attempt to write a custom HTTP server very time-consuming and impractical, considering all the tools that are already available. Another problem with writing a custom server would be security concerns – one of the biggest advantages of well-known frameworks is that they are thoroughly tested and any security vulnerabilities are quickly patched.

3.6 Webpack

Before an application can be deployed, it must first be built. “Building” is an umbrella term for many tasks that need to be done before the application is ready to be deployed. The most common build steps are compiling, linting, running tests, code minification and removing temporary files. As the project grows beyond a few source files, it can become tiresome to handle all these tasks manually as it can be time-consuming and error-prone. This is where build systems come in. The main goal of a build system is to automate all these tasks so that the developers can focus on more pressing matters. There exist many free and open-source build systems/task runners available in the Javascript ecosystem – Gulp, Grunt, Webpack and many others. In this project, we use a combination of NPM[13] scripts and Webpack[18].

Webpack is a Javascript task runner and module bundler. Its main function is to take Javascript modules used throughout the project and bundle them together into a single file that is included inside the main HTML file. Webpack makes it very easy to include Node.js modules in the frontend as they can be simply imported wherever they're needed and Webpack will handle the rest. This makes it very easy to handle frontend dependencies and it is the main reason we chose to use Webpack as our build system for the Javascript portion of the project.

3.7 Documentation

Documentation is an essential part of every larger project. While source code documenting will always be a manual task, there exist many software tools that can take the documented source code and turn it into PDF, HTML or Latex documents automatically. We use a tool called JSDoc to create the documentation.

3.7.1 JSDoc

JSDoc is a documentation generator for Javascript. JSDoc can document classes, functions, members, variables, methods and even modules. JSDoc comments have to be placed right before the code being documented and the comment has to start with /**. It is possible to use special tags starting with @ to give more information about the code being documented.

The following example is taken from a module which interacts with Graphviz. At the top of the file is a comment with a @module tag which informs JSDoc which module the file and in turn the functions belong to. The function comment uses two more tags. The @param tag tells JSDoc what parameters the function takes and what their data types are. @returns tag describes the return value. Once the documentation is written, one can use the JSDoc command line tool to turn it into HTML files that include the appropriate formatting and styles, making it easier to read. Every documented object also comes with a link to the actual source code to save time searching through the actual files.

```
/**
 * Generate image from a graph in the DOT format.
 * @module img-gen
 */

/**
 * Calls the underlying Graphviz package to
 * create a png image from a given graph.
 * @param {string} graph - graph in the DOT format
 * @param {string} engine - the graphviz engine
 * @returns Promise that resolves with a binary buffer
 */
function generate({ graph, engine, format }) {
  const command = `${engine} -T${format}`

  const childProcess = exec(command, { encoding: "binary" })

  /*
    The rest of the code is left out for brevity.
  */
}
```

Listing 3.5: JSDoc example taken from the project.

Chapter 4

Graph6

Before we describe the API and the implementation itself, we would like to describe the graph6 format that is referenced throughout the next chapters. Graph6[5] is a graph storage format developed by Brendan McKay. Graph6 is intended to encode simple graphs – undirected graphs that do not contain loops or multi edges. Converting a graph into the graph6 format yields a string of printable ASCII characters. The format is very compact and results in a much smaller footprint than e.g. an adjacency matrix from which it is derived[6].

This project relies heavily on this format. Whenever we need to pass a graph from one function to another, or from one system to another, it is done by first converting the graph in the graph6 format.

In the next two sections we describe how a graph is converted from an adjacency matrix to graph6.

4.1 Encoding Vertices

If the number of vertices is in range $[0, 62]$, 63 is added to this number and the number is converted to an ASCII character. For example, if the graph has 2 vertices the resulting number is 65 and the ASCII character is A. If the number of vertices is in range $[63, 258047]$, the number is first converted to its 18 bit binary representation. Then, the number is split into 3 groups of 6 bits, 63 is added to each of them and the numbers are converted to ASCII characters. The final step is to prepend this string with the symbol ~(tilde). For example, assume the number of vertices is 1000. The binary representation of this number in 18 bits is as follows:

$$000000 \quad 001111 \quad 101000$$

We now take each group of 6 bits and convert it back to the decimal representation, adding 63 to each:

0	15	40
63	78	103

The final step is to convert these numbers to ASCII and prepend ~(tilde):

$$? \quad N \quad g$$

The graph6 format can represent graphs that have more than 258047 vertices, but we do not consider them in this project since these graphs are simply too large to handle and too large to be effectively manipulated.

4.2 Encoding Edges

To encode graph edges, consider the adjacency matrix of the given graph and its upper right triangular matrix above the main diagonal. The algorithm traverses the triangular matrix, going column by column from top to diagonal, not including diagonal, and from left to right, and constructs a sequence of ones and zeroes.

The sequence is split into groups of 6 bits much like the number of vertices. If the length of the sequence is not divisible by 6, zeroes are added at the end. The algorithm adds 63 to each group of 6 bits and converts them to ASCII characters. The resulting graph6 is the encoded string of vertices and edges put together.

4.3 Example

Consider a graph given by this adjacency matrix:

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

This graph has 5 vertices:

$$5 + 63 = 68 \rightarrow D$$

The edge sequence is as follows:

$$(m_{1,2}, \quad m_{1,3}, \quad m_{2,3}, \quad \dots \quad m_{3,5}, \quad m_{4,5}) = (110001 \quad 1110)$$

Since the number of elements in the triangular matrix is $n*(n-1)/2 = 10$, two more zeroes are added at the end. The sequence is split into two groups of 6 bits, adding 63 to both and converting them to ASCII:

$$\begin{array}{rl} 110001 & 111000 \\ 49+63 & 56+63 \\ 112 & 119 \\ p & w \end{array}$$

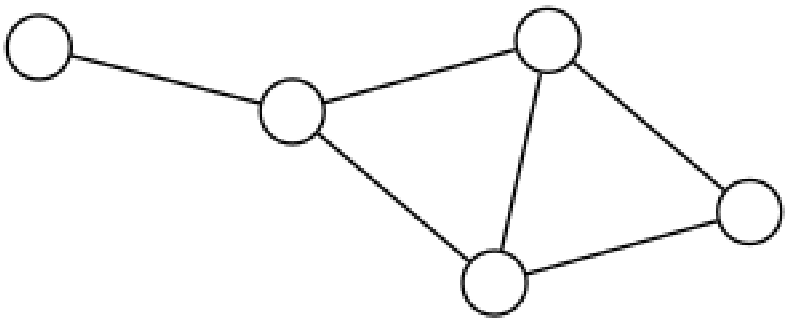

Figure 4.1: Image of a graph represented by "**Dpw**" in graph6.

The resulting graph6 representation of this graph is "**Dpw**". As we can see, this format is much more compact than the more popular formats like the adjacency matrix or the adjacency list, mostly because it sacrifices legibility for a smaller size, which helps reduce the storage requirements.

4.4 Possible Problems

The graph6 format uses only printable ASCII characters, so most of the time there should be no issues related to character encoding. One possible problem that may arise is caused by the fact that one of the printable characters is "\" (backslash) which has a special meaning in many programming languages and environments and could cause problems. It is possible that backlash together with the next character could be treated as a special character like the line feed "\n" and become a single character. We came across a similar problem while developing, where for some reason PostgreSQL escaped backslash with another backslash before inserting the graph into the database. This consequently corrupted the graph6 string. While this caused a problem, thanks to the graph6 format, any change in length results in an invalid graph6, so this particular issue was easily spotted and fixed. Special care should also be taken when handling these in Shell scripts as the characters "[]{}" and "|" are all valid graph6 characters, but all have special meaning inside Shell, especially "|"(pipe), which is the output redirection command, could cause serious problems.

Chapter 5

API Design

Now that we have selected our technology stack in chapter 3, we also need to define the HTTP API. API or application programming interface usually refers to a collection of procedures, functions or protocols of some library or a program that can be used by programmers in a certain way. In the context of a web application, API usually refers to a set of URL endpoints that can be accessed over HTTP and allow programmers to communicate with the application via the HTTP or HTTPS protocols.

There are three distinct services that we want this application to offer to the users – database, computation and images. For this reason, our API defines three separate routes all having the same prefix – /api:

/api/image	Images
/api/graph	Computation
/api/search	Database search

5.1 /api/image

This endpoint further splits into three more endpoints.

5.1.1 /api/image/list

This endpoint returns a JSON array containing file names of images for a given graph name.

Parameter	Type	Required
name	String	yes

- name – The name of the image.

Example Requests and Responses

```
GET /api/image/list?name=Petersen%20Graph

200 OK
Content-Type: application/json
[
  "Petersen_graph_01.svg",
  "Petersen_graph_02.svg",
]
```

Listing 5.1: Requesting file names of all images of the Petersen graph.

5.1.2 /api/image/static

This route returns a static graph image based on the provided file name. As described in section 6.8.2, the database stores names of some well-known graphs. Given a graph name, one can use **/api/image/list** and **/api/image/static** to view all images stored for a given graph.

Parameter	Type	Required
name	String	yes

- name – The file name of the image.

Example Requests and Responses

```
GET /api/image/static?name=Petersen_graph_01.svg

200 OK
Content-Type: image/svg+xml
```

Listing 5.2: Requesting a file “Petersen_graph_01.svg”.

5.1.3 /api/image/gen

This endpoint returns an image of a graph generated by Graphviz in the backend. The query fields are:

Parameter	Type	Required
g6	String	yes
engine	String	no
format	String	no

- g6 – This field contains the graph in the graph6 format.

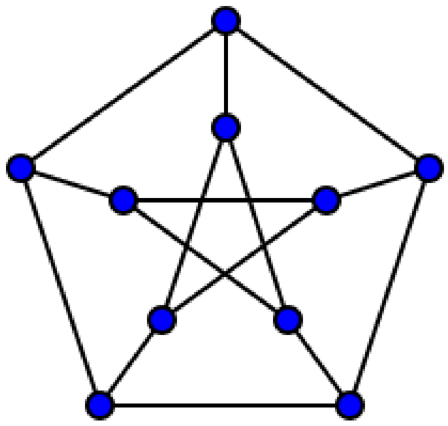

Figure 5.1: Image returned when requesting endpoint from listing 5.2.

- engine – The name of the Graphviz engine used to generate the image. The supported Graphviz engines are dot, neato, fdp, twopi and circo. Some of these engines work best for only a certain type of graphs. In general, neato and fdp seem to work best. This is an optional field, the default value is neato.

- format – The field specifies the output image file format. Three formats are supported – PNG, JPEG and SVG. This field is optional, the default value is PNG.

On success this route returns HTTP status code “200 OK” with the appropriate Content-Type. For incorrect query, which could include passing incorrect graph6 graph or an unsupported engine, this endpoint returns “400 Bad Request” with a JSON message explaining why the query was incorrect.

Requesting an image of a graph that has more than a hundred vertices results in an automatic “400 Bad Request”. The reasoning is that with graphs this big, an automated visualization generally add nothing of value. At the same time, this prevents malicious requests from requesting images of graphs with a large number of vertices.

Time limit for the image drawing process to finish is set to five seconds, which should be more than enough for most graphs based on the speed comparison in section 6.7. If the process is not finished by then, it is stopped and “408 Request Timeout” is sent back with a JSON message informing that the process timed out.

This sort of failsafe is needed as the server could be easily misused by sending requests that would take a long to complete. This could in turn radically slow down the server the application is running on, since these processes would be essentially wasting system resources. With the time limit in place, we only need to worry about attackers sending a large number of requests in a small time frame, which can be solved with rate-limiting using Nginx and is described in section 6.2.

5.1.4 Example Requests and Responses

```
GET /api/image/gen?g6=GCdenk

200 OK
Content-Type: image/png
```

Listing 5.3: Example of a correct request.

```
GET /api/image/gen?g6=GCdenk&engine=fdp&format=svg

200 OK
Content-Type: image/svg+xml
```

Listing 5.4: Example specifying the engine and the format.

```
GET /api/image/gen?g6=GCdenk&engine=unknownEngine

400 Bad Request
Content-Type: application/json
{ message: "Invalid engine" }
```

Listing 5.5: Example of an incorrect request.

5.2 /api/graph

This route is intended for requesting graph properties of a single graph. Requesting this endpoint triggers the underlying graph theory libraries. Just like with /api/image this endpoint is accessed via HTTP GET request with the appropriate query fields. This route uses two valid query formats: simple and advanced.

5.2.1 Simple Query

Parameter	Type	Required
g6	String	yes
properties	String	yes
timeLimit	Integer	no
engine	String	no

- g6 – graph in the graph6 format.
- properties – comma-separated string of property names that are to be computed.

- timeLimit – integer specifying how many seconds the computation should take in the worst case.

- engine – Name of the library that used to compute the properties. Not all libraries support all properties. This fields is optional and default value is sage. The full list is: sage (SageMath), wolfram (Wolfram Mathematica) and networkx (NetworkX).

This simple query style is best suited when requesting properties that do not require any additional arguments. For properties that take arguments, the advanced query format is required. It is also much faster to type this request by hand if need be.

For a correct query that finishes in time, this endpoint returns “200 OK” with a JSON object where each key is a property name and each value is the value of the computed property.

For incorrect queries, “400 Bad Request” is returned with a JSON message informing why the query was incorrect.

The default time limit is five seconds and the maximum time limit that can be requested is thirty seconds. If the computation does not finish in time “408 Request Timeout” is returned with a JSON message. Similar to the /api/image endpoint, this is a security measure to prevent attackers from having the server compute difficult properties on very large graphs which would, for all intents and purposes, never finish and waste resources.

Example Requests and Responses

```
GET /api/graph?g6=A_&properties=nodes,edges

200 OK
Content-Type: application/json
{
    "nodes": 2,
    "edges": 0
}
```

Listing 5.6: Example of a correct request.

```
GET /api/graph?g6=A_&
        properties=nodes,edges&engine=wolfram

200 OK
Content-Type: application/json
{
    "nodes": 2,
    "edges": 0
}
```

Listing 5.7: Example of a request specifying the engine.

```
GET /api/graph?g6=A_&
        properties=nodes,edges&engine=unknownEngine

400 Bad Request
Content-Type: application/json
{
    "message": "Invalid engine"
}
```

Listing 5.8: Example of an incorrect request.

5.2.2 Advanced Query

Some graph properties require additional arguments that the simple query format cannot handle. A more advanced format is needed to facilitate this. The advanced query defines these fields:

Parameter	Type	Required
json	String	yes
engine	String	no

- json – JSON string describing the request.
- engine – name of the graph library.

The json field value is a stringified Javascript object that has these fields:

Parameter	Type	Required
g6	String	yes
properties	Array	yes
arguments	Object	no
timeLimit	Integer	no

- g6 – graph in graph6 format.

- properties – Similar to simple query in that it defines which properties to compute, though in this case it is a Javascript array instead of a comma-separated string.
- arguments – “arguments” is an object where each of its keys is a property name. The value of this key is the arguments for the corresponding property. The argument can either be a single value or an array of values.
- timeLimit – integer specifying how many seconds the computation should take in the worst case.

Example Requests and Responses

```
GET /api/graph?json=
        { "g6": "A_", "properties": ["nodes", "edges" ] }

200 OK
Content-Type: application/json
{
    "nodes": 2,
    "edges": 2,
}
```

Listing 5.9: Example of a correct request.

```
GET /api/graph?json=
        { "g6": "A_", "properties": [ "k_regular" ],
          "arguments": { "k_regular": 10 } }

200 OK
Content-Type: application/json
{
    "k_regular": false
}
```

Listing 5.10: Example of a request specifying arguments.

```
GET /api/graph?json=
        { "g6": "A_", "properties": [ "k_regular" ],
          "arguments": { "k_regular": 10 }, "timeLimit": -20 }

400 Bad Request
Content-Type: application/json
{
    "message": "Invalid time limit"
}
```

Listing 5.11: Example of an incorrect request.

5.3 /api/search

This API endpoint facilitates communication between users and the graph database. Using a JSON format described below, similar to /api/graph, users can query the database for graphs which have certain properties. The request mimics SQL to make it easier for users to construct these requests. Some plain English example requests that can used with the API include:

- Get all graphs which have less than 10 vertices.
- Get all graphs which have radius 10 and chromatic number 6.
- Get all graphs which are not trees and have more than 15 edges.

As these examples show, we can encode arbitrary amount of conditions on the graph properties connected with a logical and operator. One of the limitations is that it is currently not possible to ask for e.g. all graphs that are either trees OR strongly regular. This issue can be solved by issuing two separate requests for all trees and all strongly regular graphs and filtering out duplicates based on their graph6 representation, which the database provides.

Similar to /api/graph advanced query, we need something more powerful than just simple query fields to encode these requests. We employ the same solution as with /api/graph – pass a JSON string describing the request.

Parameter	Type	Required
json	String	yes

The json object has these fields:

Field	Type	Required
columns	Array	no
limit	Integer	no
offset	Integer	no
format	String	no
conditions	Array	no
orderby	Array	no

- columns – Javascript array specifying which columns (graph properties) are to be returned from the database. This field is optional and if not provided, all columns are returned.
- limit – Defines the maximum number of rows that are returned. The default is 100, the maximum is 10000.
- offset – Directly maps to SQL OFFSET clause. The default value is 0.
- format – Specifies the output format of the results. The two possibilities are JSON, which is the default, and CSV. If CSV is selected the column separator is a semicolon and the header is included.

- conditions – Javascript array of conditions. Each condition is in turn an object.
- orderby – Specifies the ordering of the results. Similar to conditions, this is also an array of objects. Each object corresponds to a single SQL ORDER BY clause.

To explain how exactly conditions are defined we include a simple example. Let's say we want to encode the condition:

number of vertices is greater than 5

The resulting conditions array is:

```
[
  { "column": "nodes", "op": ">", "value": 5 }
]
```

Listing 5.12: Example Condition.

This condition maps to a simple SQL WHERE clause:

```
... WHERE nodes > 5 ...
```

The fields defined for the condition object are:

Field	Type	Required
column	String	yes
op	String	yes
value	Any	yes

- column – the name of the property that the condition relates to.
- op – the comparison operator that will be applied.
- value – the compared value.

The supported operators for each column data type are:

Data types	Operators
Integer	=, !=, >, <, >=, <=
String	=, !=
Boolean	=, !=

To expanded more on how **orderby** works, say we want to order the database results by the graph6 string in ascending order and than by the number of vertices in descending order. The orderby array would look like this:

```
[
  { "column": "g6", "order": "ASC" },
  { "column": "nodes", "order": "DESC" }
]
```

Listing 5.13: Example Ordering.

The fields defined for **orderby** objects are:

Field	Type	Required
column	String	yes
order	String	yes

- column – The database column to order by.
- order – The order of the results - ascending (ASC) or descending (DESC).

For a correct query the server sends back "200 OK" with the database results, for invalid queries "400 Bad Request" is returned with a JSON message.

Example Requests and Responses

```
GET /api/graph?json={
      "columns":["g6","nodes","edges"],
      "limit": 2
}

200 OK
Content-Type: application/json
[
    { "g6": "?", "nodes": 0, "edges": 0 },
    { "g6": "@", "nodes": 1, "edges": 0 },
]
```

Listing 5.14: Example of a correct request.

```
GET /api/graph?json={
       "columns":["g6"],
       "limit": 3,
       "conditions":[
        {"column":"edges","op":"=","value":0}
        ]
}

200 OK
Content-Type: application/json
{
    { "g6": "?" },
    { "g6": "A?" },
    { "g6": "B?" }
}
```

Listing 5.15: Example of a request specifying conditions.

Chapter 6

Implementation

In this chapter, we describe the implementation of the individual parts of the project as well as the way they work together. The project is developed with the intention to make it compatible with both Linux and Windows even though the target OS is Debian, a Linux distribution. The reason for this is not to run a production version on a Windows machine, but rather to make it easier to develop this project. I use Windows as my OS of choice so I wanted this project to be easy to develop without the need for OS virtualization, though the entire project is tested on Linux as well.

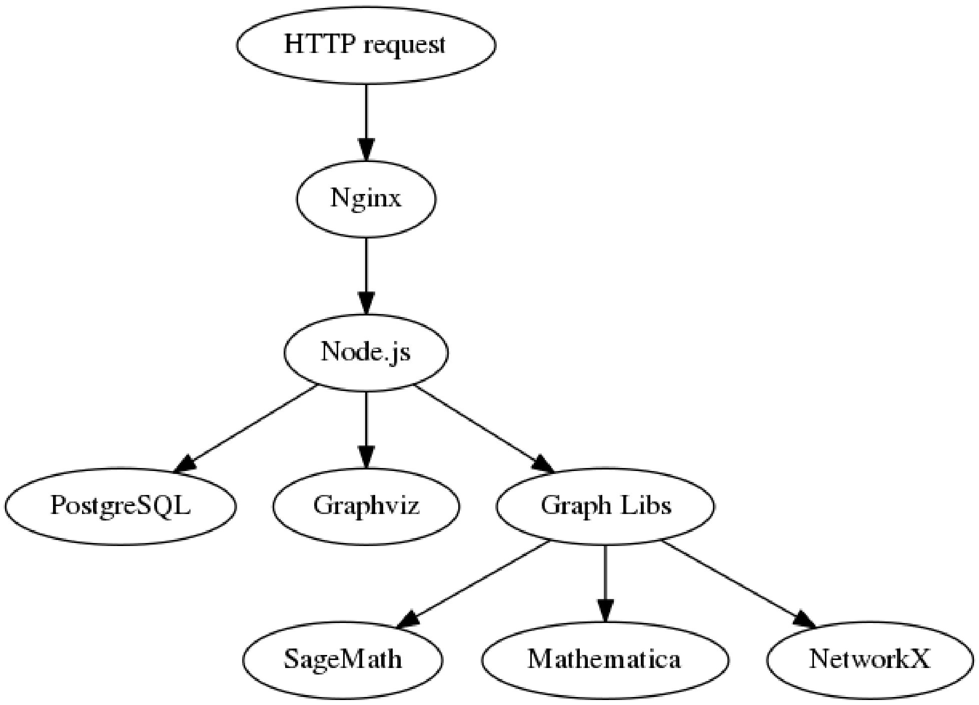

Figure 6.1: The project architecture.

```
/
├─ db_update/ ............ root directory of the "graphs" utility
│  ├─ docs/ .............. documentation
│  ├─ test/ .............. tests
│  ├─ README.md .......... readme file
│  └─ graphs ............. main entrypoint to the utility
├─ node-www/ ............. root directory of the Node.js web server
│  ├─ compute/ ........... graph theory libraries directory
│  │  ├─ networkx/ ....... NetworkX scripts
│  │  ├─ sage/ ........... SageMath web server
│  │  ├─ networkx.js ...... NetworkX driver
│  │  ├─ sage.js ......... SageMath driver
│  │  └─ wolfram.js ....... Wolfram Mathematica driver
│  ├─ db/ ................ database connection & query building
│  ├─ docs/ .............. documentation
│  ├─ images/ ............ static graph images
│  ├─ img-gen/ ........... Graphviz driver
│  ├─ log/ ............... application logger
│  ├─ public/ ............ UI static assets
│  ├─ routes/ ............ API routes handlers
│  ├─ shutdown/ .......... functions responsible for proper shutdown
│  ├─ src/ ............... UI source code
│  ├─ test/ .............. tests
│  ├─ README.md .......... readme file
│  └─ app.js ............. main entrypoint to the Node.js web server
└─ README.md ............. readme file
```

Figure 6.2: The project structure at the time of writing. Several files are left out for the sake of brevity.

6.1 Development & Deployment

We developed this project using Git[4] for version control. We found that it is the best way to share code and work on a project together with the other developers. The project consists of two main branches. The master branch is the current stable development branch that everyone has access to. Any bug fixes and minor changes go directly to that branch. The second main branch is “prod”, the production branch. Since the production build differs from the development in some details, we felt it is best to keep two separate branches. One of the differences is e.g. logging, described in section 6.10, which is configured differently on both branches.

Any changes that effect the entire project as a whole should first go to the master branch, which is then merged into the production branch. The server at graphs.felk.cvut.cz has a clone of the repository and tracks the production branch. Any larger features e.g. a new API endpoint, go on a new separate branch first, where the changes are validated with the help of unit and integration tests and then merged into master and eventually into the production branch. We always try to avoid making changes directly to production branch if we can help it, to avoid the possibility of breaking or disrupting the production server.

We considered using continuous integration to prevent having to manually restart the server each time a change is made on production branch. Though, the restart mechanism is very simple and we are at the point where the application is stable and merges happen infrequently, so we see little benefit to it at this stage.

6.2 Nginx

Nginx is configured to be the main entry point of the web server. All requests go through Nginx first before they are sent to Node.js and further to SageMath and other parts of the application. Nginx for Debian is available as a system daemon and is configured via the systemctl utility that comes with systemd init that Debian 8 uses.

Nginx configuration is kept at /etc/nginx. The main configuration file that Nginx reads is nginx.conf. It contains several configuration contexts. A context is a block of directives enclosed in curly braces:

```
http {
  sendfile on;
  tcp_nopush on;
  tcp_nodelay on;
  keepalive_timeout 65;
  types_hash_max_size 2048;
}
```

Listing 6.1: Nginx default http configuration.

The two main contexts are **events** and **http**. The events context specifies how Nginx handles incoming connections e.g. the number of workers or the processing method. The http context directives define how Nginx treats incoming HTTP and HTTPS connections. This includes the way headers are sent, keepalive settings, logging and sending static files.

6.2.1 Global Configuration

While it is possible to simply alter the nginx.conf file, it is not recommended. The usual way to add custom configuration is to create a configuration file ending with *.conf* in */etc/nginx/conf.d* directory. The nginx.conf file contains an include directive that loads all .conf files from that directory. That way all our custom configuration can be easily managed. The custom global configuration will include a single directive:

```
limit_req_zone $binary_remote_addr
                    zone=nodeLimit:1M rate=50r/s;
```

Listing 6.2: Nginx global configuration.

This directive sets up a rate limiting zone, that we will use in our server to limit the number of requests from a given IP address. The zone parameter defines the name of this limiting zone as well as the memory size to store the IP addresses. 1 megabyte is roughly equal to 16000 addresses in the binary format. If the memory runs out, Nginx will delete the oldest recorded IP addresses.

The rate parameter defines the maximum number of requests that a single IP can send. In this case it is 50 requests per second. The rate limit is set higher than it needs to be if it were just for the API itself. This is because the limit also counts towards the website, which has many resources of its own. Simply requesting graphs.felk.cvut.cz will in turn cause requests for CSS and JS files which can lead to a high number of request in a short amount of time.

6.2.2 Server Configuration

The canonical way to add a server configuration is to create a file in /etc/nginx/sites_available directory. Much like the global configuration, all files in this directory get imported into the main configuration file. If we wish to enable our server, we also need to create a symbolic link to this file in /etc/nginx/site_enabled. All servers defined in /etc/nginx/site_enabled are then considered by Nginx when deciding where to route an incoming request. The Node.js server configuration looks like this:

```
server {
    limit_req zone=nodeLimit burst=25 nodelay;
    listen 80;
    server_name localhost;
    location / {
        proxy_pass http://localhost:8080;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

Listing 6.3: Nginx server configuration.

The server block defines a new virtual server within Nginx. Next, we set up rate limiting using the limiting zone we created in the global configuration in section 6.2.1. The server listens on port 80 – the default port for all HTTP traffic. The location blocks are used to match specific routes within the server. Since we want to proxy all requests to our Node.js server we will use location / which matches every request. Inside the location block we configure the proxy. The proxy_pass directive specifies where to forward the requests. In this case it is localhost port 8080 on which the Node.js server is listening. We also tell Nginx to set X-Forwarded-For header so that the Node.js server knows the correct request IP and can log it correctly.

6.3 Node.js

When a HTTP request is processed by Nginx, it is forwarded to the internal Node.js web server that handles all the business logic. As we described in chapter 3, we use a library called Express to help us implement the server itself.

Express is basically comprised of two main concepts – middleware and route handlers. Middleware in the context of Express is a function that is executed for every request regardless of the URL, while a route handler is a function that is executed only for requests matching the given route. Each new request first passes through the middleware chain and only then is passed to the appropriate route handler.

We use the Express static middleware to serve our static assets – CSS, JS and HTML files, and also static images. Logging is done by implementing a custom middleware function that logs every request. The following is an example that logs every request:

```
app.use((request, response) => {
  // request object contains information about the HTTP request
  // response object is used to send the response
  logger.log(request)
})
```

Listing 6.4: Middleware Example.

Route handlers are used for dynamic content, which in this case is the API. If we wanted to create a new route handler for **/api/search**, this is how it could look:

```
app.get("/api/search", (request, response) => {
  // handler logic
})
```

Listing 6.5: Route Handler Example.

6.3.1 DB Connection

Since opening new database connection is an expensive process both in terms of time and memory, we must be careful how we handle the connections. If we initiated a new connection for each new user, we could face some performance issues, potentially rendering the application unusable. The most common way and probably the best way is to use a connection pool. The pool contains a certain amount of live connections that are not closed upon completion of a request but rather kept alive. When a new query arrives the connection is reused. This way, we can achieve faster response times and save resources.

6.3.2 SQL Injection

The server provides an API to request data from the database. In situations when we expose the database to the world, we must be wary of potential SQL injection attacks. SQL injections arise most often when the application builds queries by simply concatenating strings containing user input. Consider a query that is constructed in the following way:

```
query =
  "SELECT * FROM users WHERE name = '" + name + "';"
```

If a user were to pass in the string **john';DROP TABLE users; - -**, the query becomes:

```
query =
  "SELECT * FROM users
        WHERE name = 'john';DROP TABLE users;--';"
```

This query will try to find the matching user, but also drop the table immediately afterwards.

There are several ways to protect the application from an SQL injection. It can be partially mitigated by setting the correct privileges for the database role, which we explain in more detail in section 6.8.1.

It can also be mitigated by using prepared statements. Prepared statement is an SQL query as it would be executed, but the actual values are substituted with special symbols that the database recognizes as a placeholder. In PostgreSQL, the symbols are \$1,\$2,..,\$N. If we wanted to create a prepared statement for the previous query, it would look like this:

```
query =
  "SELECT * FROM users WHERE name = $1;"
```

The query is then passed separately along with the user supplied value to the database, which handles the substitution. The problem with this approach is that it is only suitable when the query is known beforehand. The nature of the search API is that the resulting query is highly irregular and dynamic, meaning prepared statements won't work in this case.

There is no simple way to solve this issue, the query must be built dynamically, though we can still pass the query and the values separately. This is one of the areas where we heavily employ unit tests and code coverage tools, described in further detail in chapter 7, to make sure that every single edge case is tested.

6.3.3 Error Handling

Some parts of the application e.g. PostgreSQL, Graphviz or SageMath can encounter unexpected errors. These may be caused by a logic error, a bug in the library, an OS error and many others. Ideally, these errors should never be propagated to the user, as that poses a security risk. The Node.js API uses rigorous error checking for all its components and never allows errors to be propagated back to the user. Whenever an error occurs it is logged to a file and a user-friendly message is sent back instead. The same is true with the Express server itself, if there is an uncaught exception thrown in one of the route handlers, only a simple error message is reported back to the user.

6.3.4 Scaling

Node.js is by design a single threaded application which could pose a challenge for CPU intensive tasks, but the server mostly deals with IO intensive tasks (PostgreSQL) or offloads the CPU heavy tasks to its child processes (Graphviz, Mathematica, NetworkX) or other services (SageMath). All of these tasks are performed asynchronously, meaning the code execution is never blocked waiting for one these tasks to finish, allowing the server to handle many simultaneous users. If we ever need to add more computing power to the server we can simply create more instances of the Node.js web server, which is what Node.js was designed for, after all.

6.4 SageMath

The SageMath library can be used in various ways. It can be used with a Jupyter notebook to create code documents similar to that of Maple or Matlab. It also comes with its own REPL (terminal interface) that can be used from the terminal. Last but not least, one can write full-fledged Python scripts and programs by including it as a Python module. The first two ways do not lend themselves to an easy automation, which is what we need to achieve. The third option, writing a Python script, is the most suitable for our needs.

The simplest approach appears to be to simply write a SageMath script and have Node.js spawn it as a new process each time it is needed. This would normally be the best choice and we use this approach for other libraries, though due to the sheer size of SageMath, it takes several seconds to load and actually start processing. This means that each time a user wants to compute anything, they will have to wait several seconds no matter how trivial the computation itself would be.

The solution to this problem is to implement a simple SageMath HTTP server that will be constantly running in the background, spawning new worker processes as needed. The worker process will be created via the **fork()** system call, which in Linux uses a copy-on-write technique, that only copies data when it is changed. Since we only use a tiny percentage of the SageMath library, the new processes should be fairly lightweight compared to the parent process.

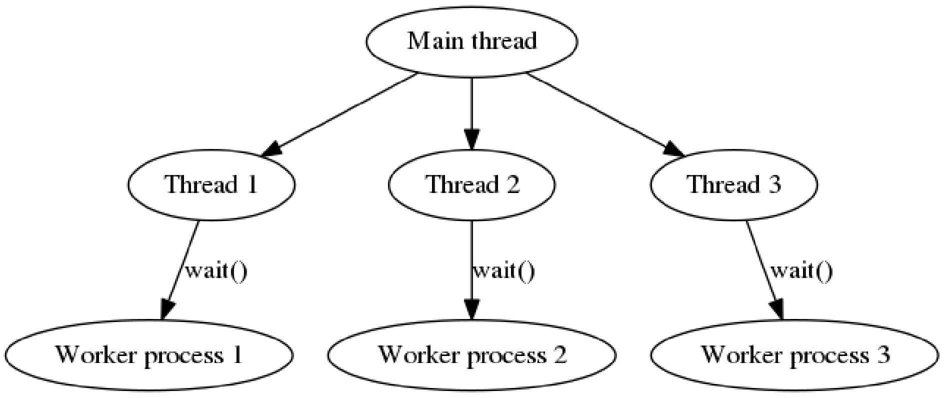

Figure 6.3: Server architecture.

When a new connection is established the server spawns a new thread that will handle the request. The thread performs request validation and if the request is correct, spawns a new worker process and waits for the process to finish. If the worker process does not finish in time, it is stopped via a signal. The results (or an error message in case of a timeout) are then sent back. The reason to spawn a thread and then spawn another process from inside the thread as opposed to just spawning a thread is as follows. Ideally, we would like to have control over how much time is spent on a single request. We do not want to expose the server to potential denial of service attacks, so we will limit the time allotted to each request. If the computation takes too long, it should be stopped. Because the SageMath library functions are essentially black boxes to us, there is no simple way to time the execution and simply stop it. Python also does not offer a way to stop threads from the outside. Another reason, why we chose to use a worker process instead of a thread is that Python does not allow two threads to run at the same time. This is done because Python objects are not thread safe and it is implemented via something called the GIL – the Global Interpreter Lock. Essentially, any thread that wants to run, needs to first acquire the GIL. This ensures that only one thread is running at any given moment and thus guarantees thread safety. Because of the GIL, we will not be able to use more than a single processor core if we limit ourselves to Python threads. To leverage the potential of multiple cores, it is necessary to use processes instead of threads.

It is now clear why we chose to use processes over threads, but then why not simply spawn a single process for each request instead of a thread and a process? We need to wait for the process to finish. Since **wait()** is a blocking operation we cannot call it in the main thread, because that would block the server itself from taking new connections, which is undesirable. For this reason, we first spawn a thread that sets up the new worker process and waits for it to finish. We mentioned previously that Python threads cannot run simultaneously, but that won't cause problems with **wait()** since its implementation explicitly releases the GIL before calling **wait()**, allowing other threads to run.

The HTTP server is based on the BaseHTTPServer module which is a low-level wrapper around the HTTP protocol. This ensures that we incur as little overhead as possible. As discussed, the main thread spawns a new thread for each new request. Its main tasks are:

- Parse an incoming request.
- Spawn a new worker process.
- Wait for the worker process to finish, alternatively stop it.
- Send results back in a HTTP response.

This is an implementation of the thread handler taken from the application:

```python
def run_task(g6, properties, time_limit, arguments={}):
    '''
    Spawns a new process and waits for ``time_limit`` seconds
    for it to finish. Sends ``SIGINT`` to it
    if the time limit expires

    Args:
      g6 (str): graph6 graph
      properties (list): properties to compute
      time_limit (int): time limit in seconds
      arguments (dict): property arguments

    Returns:
      tuple: (None, results) for success, (err, None) otherwise
    '''

    # create an inter-process pipe
    recv_end, send_end = multiprocessing.Pipe(True)

    # create a new worker process
    p = multiprocessing.Process(
        target=compute_properties, args=(
            canonical_g6, properties, arguments, send_end
        )
    )

    p.start()
    # wait for the process to finish
    p.join(time_limit)

    if p.is_alive():
        # process has not finished,
        # will be forecefully stopped
        os.kill(p.pid, signal.SIGINT)
        p.join()
        recv_end.close()
        return ("Task timed out", None)
    else:
        # process finished in time
        results = recv_end.recv()
        recv_end.close()
        return (None, results)
```

Listing 6.6: Handler Thread Implementation.

When the worker process starts, the process installs a signal handler so that it can intercept signals from the parent. The process then starts the computation, sending the results through a pipe back to the parent thread. The worker process is implemented like this:

```python
def worker(graph, properties, arguments, pipe):
  try:
    results = compute(graph, arguments)
    pipe.send(results)
  except KeyboardInterrupt: # catch SIGINT from the parent
    pass
  finally:
    pipe.close()
    sys.exit(0)
```

Listing 6.7: Worker Process Implementation.

6.5 Mathematica & NetworkX

Compared to SageMath, both Mathematica and NetworkX are much more lightweight, when it comes to memory consumption. We can thus afford to spawn a new process for each new request instead of keeping one process running that forks itself. We may consider this approach in the future, if the performance benefits outweigh the added complexity. We implemented simple scripts for both Mathematica and NetworkX that take two command line arguments: a graph in the graph6 format and a comma-separated list of properties. The scripts write the results to the standard output. The following example demonstrates how the scripts are used:

```
$ python networkx.py "GCdenk" "nodes,edges,tree,bipartite"

Standard output:
{
  "nodes": 8,
  "edges": 15,
  "tree": false,
  "bipartite": false
}
```

Listing 6.8: Example of running the NetworkX script.

Integration of the scripts with the Node.js application is straightforward. When a user requests the appropriate API endpoint, the application spawns a new Wolfram or a NetworkX process, passing it the requested properties. The script then writes the results to its standard output, which is collected by Node.js and then sent back to the user. If the process fails to finish in the given time limit, it is terminated via the SIGTERM signal.

6.6 Speed Comparison

To better understand the differences between the graph libraries we implemented in the project, we ran several tests to see how each of them performs. First, we generated test graphs with number of vertices ranging from 10 up to 500. Each batch of graphs consisted of 20 graphs with varying edge density of $0.2, 0.4, 0.6$ and 0.8. We then selected 5 graph properties which all libraries implement:

- Connected – A graph is connected if there exists a path between any two vertices in the graph.
- Tree – A graph is said to be a tree if it is connected and it does not contain any cycles.
- Eulerian – A graph is said to be Eulerian if it contains an Eulerian circuit. An Eulerian circuit is a circuit which uses each edge exactly once.
- Diameter – The graph diameter of a connected graph is the greatest distance between any pair of vertices in the given graph.
- Maximum clique – A clique is a subset of vertices of a graph in which there exists an edge between any two distinct vertices. The maximum clique is a clique with the maximum number of vertices.

Next, the total time it took each of the libraries to compute all five properties of the given 20 graphs was recorded. Only the actual computation was measured, the time to load the file containing the graph6 strings and instantiating the graphs themselves was excluded in this first test.

The table below contains the measured data. Values are shown in seconds, totaled for all 20 graphs in each category. N/A means that computation took too long and thus was not recorded.

Mathematica performed surprisingly well for the easy to compute properties: Connected, Tree and Eulerian. This could indicate that it precomputes certain properties. NetworkX performed similarly to Mathematica and SageMath came last, though it still managed to stay below half a second even for graphs with 500 vertices. The most surprising result was the graph diameter, where NetworkX failed to handle graphs larger than 150 vertices, while Mathematica and SageMath both managed to stay below one tenth of a second and below 3 seconds for graphs with 500 vertices. The Maximum clique results were much closer to each other, with SageMath performing the best, NetworkX taking about twice as long and Mathematica about four times as long.

Finally, we also measured the amount of time it takes each library parse graph6 and create a graph instance, excluding the time it takes to read the files from the disk. The values are shown in seconds, summed for 20 graphs in each category. We can see that SageMath performed the fastest, followed by NetworkX and finally Mathematica, which again might point to Mathematica precomputing certain properties before they are requested.

In conclusion, there does not appear to be a single best library that performs significantly better for all inputs and all properties as all of them seem to have their strength and weaknesses both in terms of effective algorithms and graph representation.

Vertices	SageMath	Mathematica	NetworkX
Connected			
100	0.0102	0.0005	0.0096
200	0.0373	0.0010	0.0253
300	0.0822	0.0019	0.0499
400	0.1480	0.0035	0.0858
500	0.2297	0.0047	0.1396
Tree			
100	0.0100	0.0001	0.0014
200	0.0363	0.0001	0.0027
300	0.0812	0.0002	0.0040
400	0.1472	0.0002	0.0055
500	0.2321	0.0002	0.0071
Eulerian			
100	0.0106	0.0001	0.0003
200	0.0370	0.0001	0.0004
300	0.0914	0.0001	0.0004
400	0.1564	0.0001	0.0004
500	0.2307	0.0002	0.0004
Diameter			
150	0.0853	0.0717	19.3123
200	0.1580	0.1641	42.7813
300	0.4055	0.5313	N/A
400	0.8313	1.2406	N/A
500	1.4744	2.3556	N/A
Maximal clique			
80	0.1206	0.2680	4.9830
90	0.1820	0.3756	6.4687
100	0.2513	0.9985	8.2554
125	1.8990	7.7603	14.0859
150	11.9653	43.2370	22.2939

Table 6.1: Speed comparison for selected properties. Data shown in seconds. The leftmost column shows the number of vertices.

Vertices	SageMath	Mathematica	NetworkX
100	0.1307	0.7062	0.2121
200	0.5158	1.7343	0.8047
300	1.2019	3.7573	1.9317
400	2.2109	6.7374	3.4127
500	3.5222	10.5706	5.8074

Table 6.2: Speed comparison of graph instantiation. Data shown in seconds.

6.7 Graphviz

To produce graph images, Node.js spawns Graphviz child processes. Graphviz on Debian comes with all of its engines, described in chapter 2, available directly from the command line. Given a file with a graph encoded in the DOT language, a PNG image is produced with the following command:

```
$ neato -Tpng < graph.dot > graph.png
```

As we can see, the Graphviz engines are capable of taking a graph from **stdin** and output the generated image to **stdout**. This is very important, since to create an image it is not necessary to interact with file system at all, meaning the entire operation is only bottlenecked by the speed of the actual image creation.

To call a given engine from Node.js, the application uses the **child_process** module that can spawn new child process. When a request to a draw an image is received, Node.js spawns the selected engine as a new process and pipes the the requested graph to its **stdin**. It then waits for its **stdout** that is then sent back.

Because of the possibility for misuse, the function handling this operation implements a timeout. It is not unlikely someone could try to draw a graph with thousands of vertices and edges. Drawing such a graph could take a very long time and waste a lot of resources. For this reason, if the Graphviz engine does not exit in a reasonable amount of time, it is stopped via a signal.

To better understand how each of the Graphviz engines scale with the amount of vertices and edges, we performed a couple tests. Each test consisted of generating certain number of random graphs (between 30 and 50) with a fixed number of vertices and varying edge density. The graphs were passed to the engines and the time it took to draw the graphs was recorded. The total time was measured with the Linux time command:

```
$ time (while IFS="" read -r line ||
        [[ -n "$line" ]]; do
    echo "$line" | neato -Tsvg > /dev/null
  done < graphs.g6)
```

Listing 6.9: Timing Graphviz Engines.

The total amount of time for all graphs was measured and then averaged across the number of graphs. Figures below show the times for each engine and different edge densities.

Circo was consistently the slowest of the engines and took about 18 seconds to draw a graph with 30 vertices and 0.8 edge density. Neato and twopi on the other hand could handle graphs with 1000 vertices in less time – about 10 seconds. It is unsurprising that circo, and to an extent dot, turned out to be significantly slower than the rest. These engines are not intended to draw generic undirected graphs, however what was surprising, is how much slower fdp performed compared to neato. Both engines are supposedly well-suited for tackling undirected graphs and both use a spring model so it is unclear why such discrepancy occurs. Ultimately, what these figures have shown, is that the time to draw a graph can vary wildly depending on which engine is used and that the decision to use neato as the default engine is justified.

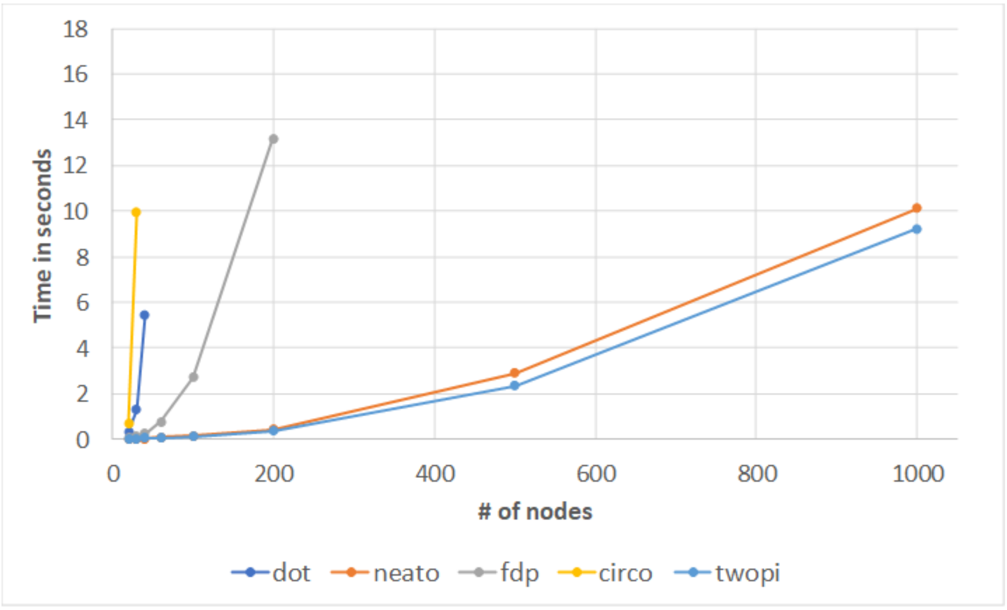

Figure 6.4: Graphs with 60% Edge Density.

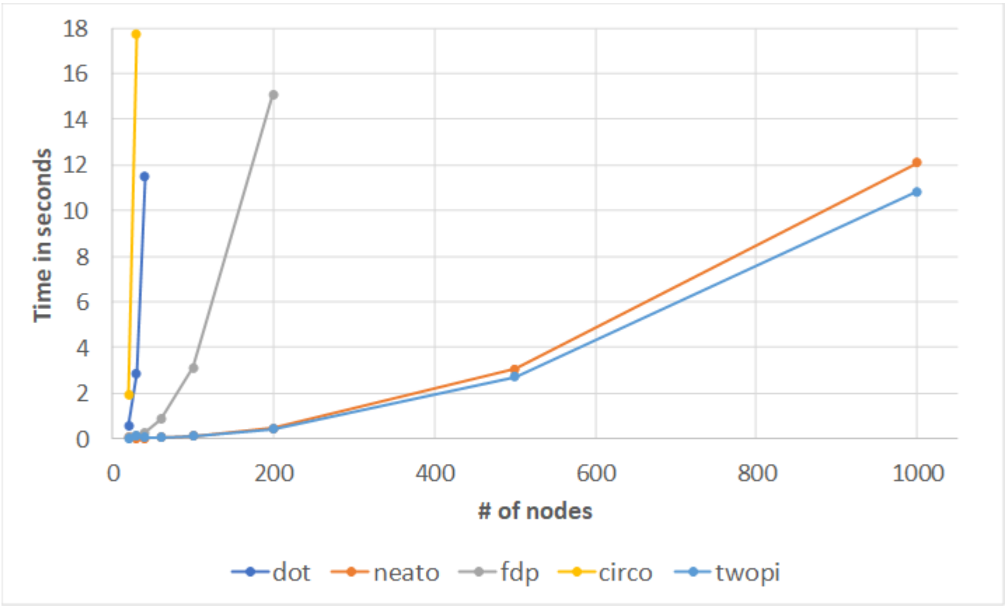

Figure 6.5: Graphs with 80% Edge Density.

6.8 Database Design

The database of choice is PostgreSQL version 9.6. It is configured to run as a service, listening for connections on localhost, port 5432. The production database is named “graphs”. It is accessed both from the application and manually through *ssh* for maintenance. The database defines a group of users (roles) that can access the database, each with a different set of privileges.

6.8.1 Roles

It is generally not advisable to work with the database under the super user, unless it is required. For this reason, the database contains three roles that can connect to it:

- Read-only user – This user can run only SELECT statements on the database “graphs” in the public schema.
- Read+Insert – This user can run SELECT and, in addition to it, can also perform INSERT statements.
- Super user – Super user has all database privileges, used for offline administration e.g. adding new graphs.

The ready-only user is used by the Node.js server when users request resources from the database. Since we do not allow users to actively alter the database in any way, there is no need for any extra privileges besides SELECT. While this does not guarantee that the application is safe from attacks, it at least makes it more difficult.

All newly created roles inherit from the PUBLIC role. The role grants them connect privileges to all databases and usage and create privileges to all schemas including PostgreSQL schemas like the **information_schema** or **pg_catalog** which contain database metadata. It is therefore necessary to revoke these privileges when creating new users.

The read-only user is created like this:

```
CREATE ROLE user LOGIN ENCRYPTED PASSWORD 'pwd';
GRANT CONNECT ON DATABASE graphs TO user;
GRANT USAGE ON SCHEMA public TO user;
GRANT SELECT ON ALL TABLES IN SCHEMA public to user;
GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO user;
GRANT USAGE ON ALL SEQUENCES IN SCHEMA public to user;
```

Listing 6.10: Creating a read-only role.

The user can only connect to a single database, can only use the public schema and can only perform SELECT statements. We also need to grant select privileges on sequences for tables with auto incrementing columns.

The select + insert user is largely similar, it includes INSERT privileges on tables in addition to all of the above. This role is used when users request /api/graph which computes graph properties. Before any computation is done, it first checks the database to see if the graph is already stored. If it is not, it is then inserted.

The superuser is used for manual maintenance only and is thus never exposed to the outside world via the web server running on graphs.felk.cvut.cz.

6.8.2 Schema

The database schema that directly relates to my thesis contains two tables.

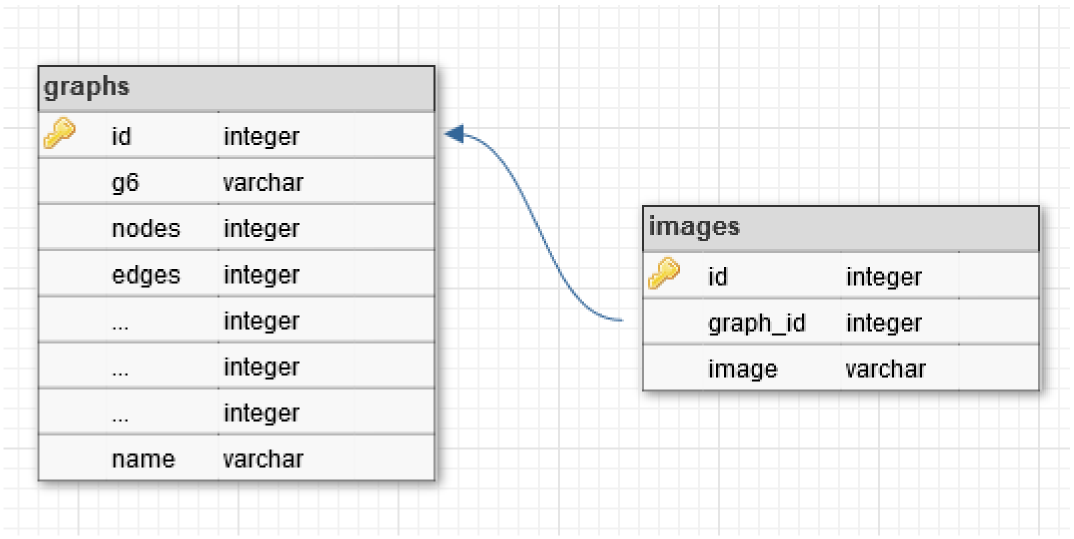

Figure 6.6: Database schema.

Table *graphs*

Table *graphs* stores the graphs. Each rows corresponds to a single graph. Each row contains the graph6 representation, the graph properties and, if the graph is well-known, its name. All of the columns are indexed to speed up the search. The graph6 column contains a UNIQUE constraint to prevent accidental insertion of a graph with same graph6 string. Column *name* also has a UNIQUE constraint since no two graphs should have the same name. The columns holding graph properties are nullable. For a given graph, NULL signifies that the given property has not yet been computed, since not all properties can be computed in a reasonable amount of time.

Some graph properties can take on values that cannot be easily represented by PostgreSQL. Some properties like radius or diameter are defined as positive infinity for certain graphs. We cannot use NULL to denote these values, since that would lead to ambiguity. We would not know whether the value is unknown or just undefined. Fortunately, all of the aforementioned properties take on only non-negative integers, so we can use negative integers to denote these special states. In this case, we use −1.

Table *images*

Table *images* holds the information about the static images that can be displayed for certain graphs. The table is in **1:n** relation to the table *graphs*, since a single graph can potentially have several images. The column graph_id is the foreign key referencing the id of the given graph in the *graphs* table. The column *image* in table *images* holds the name of the image file stored on the server.

6.9 Database Administration

The graph database is intended to be constantly growing in size, both in terms of the number of graphs and also the number of computed properties. To automate this, we created a command line utility that aids in managing the database.

The utility is named “graphs” and it is written in Bash and Python. Bash handles validating the command line arguments and calling the underlying Python scripts, while Python handles computing graph properties with the help of SageMath and interacts with the database.

The utility includes a simple Makefile that places a symbolic link to the main script into a system directory that is included in the PATH environment variable (/usr/local/bin on Debian). Therefore, the utility can be called from any directory, making it easier to use. The utility is capable of three tasks, described in more detail in sections 6.9.1, 6.9.2 and 6.9.3:

- Displaying the current state of the database – The utility displays in a simple table which properties are computed and which are not.
- Importing new graphs – The utility is capable of adding new graphs to the database from a file. The utility ensures that no isomorphic graphs are added. The isomorphism check was kindly provided by Herbert Ullrich as a part of his thesis.
- Computing missing properties – The utility can be instructed to start computing given graph properties, periodically updating the database.

6.9.1 Displaying the current state

The following command displays the current state of the database.

```
$ graphs -s [-m] [PROPERTY]...
```

Executing **graphs -s** displays the status of all properties configured in the utility. Properties in the output shown in green are those that are computed for all graphs. Properties in red are those that do not yet exist in the database. Properties that are not complete (the corresponding database column contains NULL values) have a percentage next to them indicating the number of graphs already computed (the number of non-NULL values in the corresponding database column).

If one only wants to see certain graph properties, they can be supplied as additional command line arguments following the **-s** switch.

To display only the missing properties, one can use **graphs -s -m** which excludes properties that have been computed for all graphs in the database.

6.9.2 Importing New Graphs

Another task that the utility can be used for is a mass import of new graphs. The only thing that is required, is a graph6 file containing a single graph on each line. It can be invoked like this:

```
$ graphs -g FILE
```

Here, the utility will first compare graphs in the given file with those currently in the database and filter out those that are isomorphic to them. The graphs are then imported into the database. To speed up the import, the utility uses the COPY command instead of the usual and much slower INSERT. The utility also drops the index on the graph6 database column before importing and recreates it after the import. During testing, we were able to reach about 10000 rows inserted per second with the COPY command, making it a relatively quick way to import graphs, though I imagine the speed will depend heavily on the storage medium used and not just the database itself.

6.9.3 Computing Missing Values

The most useful aspect of the program is the ability to compute missing graph properties.

```
$ graphs [-b BATCHSIZE] [PROPERTY]...
```

We specify which graph properties to compute. The utility then finds the graphs that have missing values and start computing using the SageMath library. We can also specify the batch size. The batch size tells the program how many values for each property it should compute before updating the database. This way, if the utility encounters a graph that takes too long to process, the utility can be stopped and the previously computed values will not be lost.

The utility can create columns that do not yet exist an also create indexes. Critical sections are executed in a transaction to prevent corrupting the database.

For properties with linear complexity the bottleneck is the database itself. In that case, updating takes longer than computing the values. PostgreSQL is known for slower updates, because it is implemented as INSERT + DELETE. We managed to reach several thousand updates per second. For properties that have exponential complexity, the slow nature of UPDATE stops being a real concern, especially for larger graphs. Even on smaller graphs though, some property calculations would take hours to finish e.g. the graph treewidth, and some would simply not finish at all, getting completely stuck on certain graphs e.g. the graph genus. It is unclear whether it is simply because the method is inefficient or because of a bug in the library.

Overall, the "graphs" utility has proven to be very useful in eliminating many tedious tasks that would otherwise take a very long time and would be prone to human error. At the same time, the utility does not require any special knowledge of the underlying system, meaning that anyone with a basic understanding of the Linux terminal can learn it and use it efficiently in a matter of minutes.

6.10 Logging

As described previously, the Node.js server uses Winston for logging. The logging itself is twofold. First, all logs log to the console by default. This is mostly used during development for debugging purposes. Second, on the production branch, all logs are logged to a file as well in case they are ever needed in the future. The canonical way to do logging on most mainstream Linux distributions is to log to **/var/log**.

Logging to a single file is not usually recommended, unless the application produces very few logs. We employ a technique known as log rotation. If the log file grows too large, it is renamed, usually by appending a number to it, and optionally compressed. A new file takes its place and the cycle continues until it also becomes too large. The oldest log files are then deleted.

Chapter 7

Testing

Testing, in the context of software development, refers to verifying the correctness of the source code. In other words, tests make sure that the code produces the expected results, while letting us know when it does not. Testing is an incredibly important part of every larger software project as it can discover logical errors in the code that would otherwise go unnoticed, possibly becoming a larger issue in the future. Tests also help us be more confident that the code does what it is supposed to do. Passing tests do not guarantee a bug-free code, though it is better than having no tests at all. Another big advantage of writing tests is that it forces us to write code in such a way that lends itself to easy testing. This usually results in a loosely-coupled and modular code. Last but not least, tests often act as a sort of a documentation on its own and can help developers understand what a specific part of the code is supposed to achieve. For Node.js, we use Mocha as our unit testing framework and chai as our assertion library. For Python, we use the native unittest module.

7.1 Unit Testing

Unit testing refers to testing a single unit. A unit is a small piece of self-contained functionality that can be tested independently. It could be a function, a method, a class or even an entire module or a collection of functions. The point of unit testing is to take each unit and test it in a self-contained environment completely separate from the rest of the code base. This can sometimes be difficult to achieve because several components could rely heavily on each other, making testing one of them difficult. This can oftentimes be solved by employing strategies like Inversion of Control. The reasoning behind testing a self-contained piece of code is that, if we want to have an application that behaves as expected and has a minimum amount of unexpected behavior, we first need to assert that all units behave as expected without any interference. If the modules and classes have logical error themselves, the entire program cannot work correctly.

In the context of this project, a unit could be a module that handles routing HTTP requests, a function that validates the HTTP query parameters or a class handling graph6 parsing.

7.2 Integration Testing

Integration testing refers to taking a previously unit tested group of modules and testing whether they behave correctly as a whole while interacting with one another. This step is very important, as many unseen problems could arise from running several components together.

In the context of this project, an integration test could be a test that verifies that requesting a certain resource from the HTTP server returns the expected data. For example, requesting several graphs from the database should actually return those graphs and set the correct headers as well. This verifies that the route handler, the request validation and database driver can all work together.

7.3 Mocha & Chai

Mocha is a Javascript unit testing framework running on Node.js that provides a simple command line interface to run tests. While Mocha can run our tests and report how many passed and how many failed, we will still need an assertion library. Assertion libraries make it easier to write unit tests by providing a set of methods to make assertion about our code, like comparing that two variables are equal. The following is a very simple unit test using Mocha and Chai:

```
const expect = require("chai").expect
const Matrix = require("../../utils/matrix")

describe("utils.Matrix class", function () {
  describe("Matrix.toGraph6()", function () {
    it("should convert matrix to graph6", function () {
      expect((new Matrix([])).toGraph6()).to.equal("?")
      expect((new Matrix([[0]])).toGraph6()).to.equal("@")
    })
  })
})
```

Listing 7.1: Unit test example testing conversion from an adjacency matrix to graph6.

The functions **describe()** and **it()** are provided by Mocha and they help us describe the feature we're testing and what the expected behavior of said feature is supposed to be. This description will be displayed next to the tests when we run Mocha. Chai provides the **expect()** function that implements many useful methods to do comparisons and checks e.g. checking for equality, null or data types. If the assertion fails **expect()** throws an **AssertionError** that signals to Mocha that the particular test has failed.

```
const expect = require("chai").expect
const compute = require("../../compute/networkx")
const config = require("../../compute/config")

process.env.NODE_ENV = "test"

describe("networkX driver", function () {
  it('should return correct values', async function () {
    this.timeout(5000)
    const result = await compute("A?", ["nodes", "edges"], 5)

    expect(result).to.deep.equal({ "nodes": 2, "edges": 0 })
  })

  it("should reject for time outs", async function () {
    this.timeout(5000)

    let thrown = false
    try {
      await compute("GCdenk", config.networkx.properties, 0.1)
    } catch(e) {
      thrown = true
      expect(e).to.equal("Process timed out")
    }

    expect(thrown).to.equal(true)
  })
})
```

Listing 7.2: Integration test example testing the NetworkX driver.

7.4 Code Coverage

While writing tests is undoubtedly a very useful thing, it is very easy to get a false sense of security. If the tests are not thorough enough and do not test every condition, then they are not very useful. If one writes a unit test for a function that contains a dozen branches, but the unit test only runs two of them and skips the rest because it is incomplete, the test will show up as passing, making us believe the function is correct. In reality, the function is only tested to be correct for a small amount of test cases and might break on untested inputs. The unit tests themselves do not have any built-in mechanism to gauge whether we actually tested every single code path there was to test. For that we need a tool that measures code coverage.

Code coverage tools use techniques that trace the program execution and remember which statements and branches were or were not executed. Using this tool, we can obtain a useful metric that tells us how many percent of our code was run in a unit test. Ideally, we would want 100% of our code tested but, even that does not guarantee that our code is correct, though generally a higher code coverage is preferred.

7.4.1 Istanbul

Istanbul is a Node.js code coverage tool and a command line utility intended to measure code coverage of Javascript applications. Istanbul works very well with popular unit testing frameworks such as Mocha which is the reason we are using it in this project. Istanbul supports command line reporting or it can also generate an HTML report. Thanks to its interoperability with Mocha, we can measure code coverage and run unit tests at the same time. The generated report shows the total coverage of each module imported during the run of the unit tests. Modules can be expanded to show the exact lines in the source code that were skipped.

7.4.2 Coverage.py

Coverage.py is a Python package and a tool for measuring code coverage of Python programs. Coverage.py uses the trace function invoked by the Python interpreter for each executed line to find out which lines were executed and which ones were skipped. Similar process is applied for branches. The tool by default reports to the terminal but just like with Istanbul there is a possibility to generate a more detailed HTML report. Coverage.py also works natively with the Python unittest module, which we use to test the Python modules in this project.

Chapter 8
Conclusion

In this thesis, we have presented and implemented graph database fundamental services. We have presented a database capable of storing large number of graphs and capable of effective search. We have also implemented a graph visualization service using Graphviz and compared the speed of the Graphviz engines. Next, we have implemented an online computation service using SageMath, Wolfram Mathematica and NetworkX and compared their speed on selected graph properties. Furthermore, we developed a command line utility to easily manage the database. The utility is used to import new graphs and compute values of graph properties. Finally, we designed an API and a web server that makes these services accessible through the HTTP protocol.

The full application is accessible at `graphs.felk.cvut.cz`. One can use it by directly accessing the API or via a user interface developed by Sergej Kurbanov[21]. The user interface integrates database search, graph visualization and online computation.

8.1 Possible Improvements

The project described in this thesis is a pilot project and there are many improvements and features that could be added in future iterations of the project. Firstly, we integrated three graph theory libraries into the application, but there exist many more. It is worth to consider them as they might implement additional graph properties which the project does not offer at this point. Secondly, at the time of writing this thesis, the machine which the project runs on is relatively underpowered with just 1GB of RAM and 1 CPU core. As we add more graphs the database memory requirements will grow. At the same time, some parts of the application are designed to be multi-threaded. Thus, migrating the application to a more powerful machine should be considered. Finally, the application uses the graph6 format to store graphs. Some graphs, however, are more efficiently stored using sparse6[5], a format intended for large sparse graphs. Using sparse6 to represent certain graphs could potentially lower memory requirements and speed up the application.

Appendix A
Contents of the included CD

The current version of the project can be found at `https://gitlab.fel.cvut.cz/graphs/development`. The project structure is described in figure 6.2.

```
/
├─ thesis.pdf ............. digital version of the thesis
└─ graphs/ ................ git repository of the project
```

Appendix B

Bibliography

[1] Ecma official website. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[2] Encyclopedia of graphs. http://atlas.gregas.eu/.

[3] Express official website. https://expressjs.com/.

[4] Git official website. https://git-scm.com/.

[5] Graph6 & sparse6 official website. https://users.cecs.anu.edu.au/~bdm/data/formats.html.

[6] Graph6 format description. https://users.cecs.anu.edu.au/~bdm/data/formats.txt.

[7] Graphviz official website. https://www.graphviz.org/.

[8] House of graphs. https://hog.grinvin.org/.

[9] Networkx official documentation. https://networkx.github.io/documentation/stable/.

[10] Networkx wikipedia article. https://en.wikipedia.org/wiki/NetworkX.

[11] Nginx official website. https://nginx.org/en/.

[12] Node.js official website. https://nodejs.org/en/.

[13] Npm official website. https://www.npmjs.com/.

[14] Number of non-isomorphic graphs. https://oeis.org/A000088.

[15] Python official documentation. https://docs.python.org/2/index.html.

[16] Sagemath official website. http://www.sagemath.org/.

[17] Sagemath wikipedia article. https://en.wikipedia.org/wiki/SageMath.

[18] Webpack official website. `https://webpack.js.org/`.

[19] Wolfram mathematica documentation. `http://reference.wolfram.com/language/`.

[20] Wolfram mathematica wikipedia article. `https://en.wikipedia.org/wiki/Wolfram_Mathematica`.

[21] Sergej Kurbanov. *Graph Database User Interface*. Bachelor thesis, Czech Technical University in Prague, 2018.

[22] Herbert Ullrich. *User Extensible Graph Database*. Bachelor thesis, Czech Technical University in Prague, 2018.

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Roun Tomáš**	Personal ID number:	**457086**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Branch of study:	**Computer and Information Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Graph Database Fundamental Services

Bachelor’s thesis title in Czech:

Základní služby grafové databáze

Guidelines:

Design and implement fundamental services offered by server graphs.felk.cvut.cz. The goal of the services is to support incremental batch filling of the database, database querying based on graph properties specified by the user, effective computation of properties of graphs in large collections.
Explore the possibility of calculating graph properties by external libraries/programs and utilize a selected one in your application. The database on the server will grow over time, strive for maximum modularity of your application, to enable easy later computation of additional graph properties and their recording in the database. Consider graph properties of various classes - numerical, logical, vector with variable length etc.
Your application is to be accessible through the web interface created by other authors. Provide an appropriate programmer documentation of your project.
The number of graphs in the database is expected to be in order of at least tens of millions. Identify those database tasks which might be slow or extremely slow due to the databaze size. If it is possible, suggest and investigate methods which might increase the speed of excessively time consuming calculations.

Bibliography / sources:

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms, 3rd ed., MIT Press, 2009
[2] J. Matoušek, J. Nešetřil: Kapitoly z diskrétní matematiky, Karolinum, 2010
[3] R. Sedgewick: Algorithms in C Part 5: Graph Algorithms (3rd Edition), Addison-Wesley Professional, 2002
[4] J. Demel: Grafy a jejich aplikace, Praha, Academia, 2002

Name and workplace of bachelor’s thesis supervisor:

RNDr. Marko Genyk-Berezovskyj, Department of Cybernetics, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **12.01.2018** Deadline for bachelor thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

RNDr. Marko Genyk-Berezovskyj	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature