Tool development for visualising design strategies and their ecosystem services for sustainable buildings

Vincent Ertner

5965411

Submitted in partial satisfaction of the requirements for the degree of

MSc Applied Data Science

Department of Information and Computing Sciences

Utrecht University

Supervisors: Katharina Hecht & Jaco Appelman

Second supervisor: Matthieu Brinkhuis

01-07-2022

Abstract

The effects of climate change and the high rate of urbanisation require cities to be redesigned. Buildings can implement sustainable design strategies to generate and provide ecosystem services. With the data gathered from such buildings worldwide, this project presents a tool to visualise this data. The tool is built with the micro web framework Flask and deployed to the web using Heroku. The website implements an interactive table and visualises the buildings and design strategies as nodes using Neo4j and Neovis. Furthermore, the data is kept up-to-date using an executable file extracting the data from Google Sheets and adding this to a Neo4j database.

Keywords— Sustainability, Graph Database, Graph Theory, Neo4j, Flask, Dash, Neovis

Table of Contents

List of Figures

Chapter 1
Introduction

Climate change and global biodiversity loss need to be addressed urgently since human survival depends on the various organisms that live on the planet (Zari, 2015). Changes in biodiversity impact ecosystem services, global food production and human health (Norberg et al., 2012). Since there is a rise in people living in cities and land around cities is used as a ground to build on, the relatively small area of land is polluted and high demanding in living usage. Since this trend causes loss of available habitat, it results in soil degradation, loss of ecosystem services, and loss of biodiversity (Zari, 2015). Furthermore, because of the layout of most cities, the emission of greenhouse gasses is high since cities are built for transporting people, resulting in air pollution. Newly built buildings must consider implementing design strategies to restore biodiversity and the degraded urban ecosystems and keep their ecological footprint small; already existing buildings can implement small-scale developments to improve. Examples include ecologically designed urban landscapes, using ecological processes to remediate pollution or degradation of ecosystems, and ecologically engineered walls to provide additional habitat (Zari, 2015). Rather than just minimising energy or water use, or the emission of pollutants, buildings can aim for regenerative designs to produce quantifiable ecological and social health outcomes (Zari and Hecht, 2020a).

The scope of this research project is graph visualisation and graph theory about ecosystem services in which design strategies fall. Buildings implement these design strategies to create an ecosystem resulting in sustainable buildings. This thesis project collaborates with the faculty of science, department of biology, at Utrecht University (UU) and Real Estate & Campus at the UU. The project's scope was determined together with Katharina Hecht, a PhD candidate focussing on ecology and sustainability in buildings at UU and supervisor of this thesis. The diagram Hecht created along with Pedersen Zari aims to provide information on integrating ecosystem services into buildings (Zari and Hecht, 2020a).

Architects and organisations responsible for developing buildings need insight into the industrial feasibility and ecological appropriateness of existing sustainable building designs worldwide since these designs generate ecosystem services. This thesis project aims to answer the following research question:

> *Can we design an interactive and adaptive tool that links ecosystem services to building designs and case studies?*

This tool will support building developers' decision-making toward more sustainable buildings. Interactivity will be implemented so the user can interact with the graph's different ecosystem services and building structures. Furthermore, the visualisation will be adaptive; admins need to be able to add or delete data from the graph; this way, the graph will stay up to date, and innovations can be easily added.

Visualising the nodes in the network clearly for the reader will be beneficial to gaining insights into the different design strategies that can be implemented in similar buildings, which is beneficial for people active in ecological sciences and the building industry.

A few steps had to be taken to develop this tool. First, a database of buildings and their sustainable designs had to be linked to the ecosystem services these designs generate. Then, the data was exported to a graph database (Neo4j) by adding the buildings and design strategies as nodes. A Python script was used to extract the nodes from the database, and then these nodes were added to the graph database using the query language Cypher. An online environment (Google Sheets) was used to let the admin update the sheet and, therefore, update the final graph in the tool.

This paper starts with a literature review covering background knowledge about ecosystem services and the different sustainable design implementations, followed by a more technical description of graph theory. Next, the steps regarding the tool development are described extensively in the methods. In this section, the data will be described, along with the code written to generate the graph using Neo4j and create the tool using Flask. The tool will be discussed in the results section, and the insights gained by applying graph theory on the final graph. The discussion provides room to reflect on the limitations of the research and what potential future research on the topic could entail; moreover, it will state the main findings of this research project. Finally, the research question is answered in the conclusion.

Chapter 2

Literature review

2.1 Climate change

2.1.1 Urbanisation

In the 20th century, the number of people living in cities increased rapidly. Where 15% of humans lived in urban areas at the beginning of the 20th century, around 50% of humanity across the globe lived in cities in the year 2000 (Doughty and Hammond, 2004). This trend continues today, with 55% of the world's population, 4.2 billion people, living in urban areas. At the beginning of this century, 35 cities worldwide populated over 5 million people, and over a hundred cities populated more than one million (Doughty and Hammond, 2004). Having this amount of people living on a relatively small portion of an area takes a toll on the land and its natural resources. It causes loss of available habitat, resulting in soil degradation, loss of ecosystem services, and loss of biodiversity (Zari, 2015). Transportation systems throughout the cities cause air pollution; moreover, the traffic jams result in poor air quality (Doughty and Hammond, 2004). In addition, burning fossil fuels releases greenhouse gasses that change the global climate (Grant, 2012). Furthermore, energy consumption and water use, for example, are considerably higher in cities than in rural areas.

2.1.2 Ecosystem services

To maintain a living on this planet, people are reliant on the services that planet Earth supplies: ecosystem services (Grant, 2012). These services include clean air, water, and food. These ecosystem services can be categorised into supporting, cultural, provisioning, and regulating services (Grant, 2012). Supporting services include soil formation and pollination; cultural services include recreation and the provision of aesthetic features; provisioning services include food production and water supply; regulating services include all regulations of climate extremes, like heat waves and floods (Wratten et al., 2013). Because of the effects of urbanisation, cities should maintain biodiversity for functional ecosystems (Wratten et al., 2013).

2.1.3 Sustainable buildings

Especially cities can adjust to embrace these ecosystem services by restoring the natural environment. Newly built buildings must consider implementing design strategies to restore biodiversity and the degraded urban ecosystems and keep their ecological footprint small; already existing buildings can implement small-scale developments to improve. Buildings can have more functionalities, for example, by providing food and wildlife habitat and reusing energy and water (Grant, 2012). Green roofs enhance energy efficiency since it prevents solar heat from passing through and have better thermal performance (Abdellatif and Al-Shamma'a, 2015).

2.2 Database management

A way to store information organised is in a database, a collection of related files that are usually linked to each other. A database management system is a set of software programs which allows for creating, editing and updating data in the database files and retrieving them (Bhatia and Bhatia, 2014). A database can have different forms, each providing a particular goal. A hierarchical database model arranges the data hierarchically. The relationships between the data are thought of in terms of children and parents, namely, links between parents and children, and children can not have more than one parent. Network databases are similar to the hierarchical model; however, the children can have multiple parents, supporting many-to-many relationships (Bhatia and Bhatia, 2014). Often these databases are relational, meaning the data is stored in two-dimensional tables consisting of rows and columns. Every table needs a unique column value, a primary key, to separate the row entries and eliminate redundancy (Bhatia and Bhatia, 2014).

2.2.1 ER models

Before creating a database, it is recommended to create an Entity-Relationship (ER) model. An ER model provides a clear visual representation of what information is used in the database and how data is connected (Harrington, 2009). An ER model consists of all the entities present in the stream of data and the relationships between them. An entity is something that exists, for example, objects or people. Each entity consists of attributes, properties, and one unique value that uniquely identifies the specific entity from the rest, the primary key (Gordon, 2017). Entities are linked to each other based on the same attribute values, so-called foreign keys. For example, a person that works for a company can be linked by the company ID that is present as a field in the employee table and is the primary key of the company table. In this example, the foreign key from the child table employees contains the primary key of the parent table company. The relationship between these two tables describes their connection; in this case, an employee "works for" a company.

In an ER model, entities are rectangles, relationships are diamond-shaped, and the

relationships are lines between them. Often at the end of these relationships, its cardinality is stated to understand if an entity can be linked to another entity multiple times or just once (Harrington, 2009). A company can have multiple offices but has one and only one headquarter. The different cardinalities are one-to-one, one-to-many, and many-to-many.

2.2.2 Relational Databases

The most popular database structure is a relational database. In a relational database, the data is organised in rows and columns (Harrington, 2009). The rows represent the collection of different data entries; each entity is listed on a different row. The columns consist of the attributes of an entity and are the same for every row. Each table represents an entity type; for example, data about employees and companies are listed in two separate tables.

Database Access Language SQL

Once the database skeleton is created based on the ER model, it is time to add/insert data into the database. There are a few database access languages available; however, structured query language (SQL) is the most popular. Specific data can be retrieved by sending queries to the database management system.

A few statements are used to extract data from a relational database using SQL. The SELECT statement can be combined with other statements to retrieve specific data from a table (Harrington, 2003). The table name is stated after the FROM statement to specify from which table the data needs to be selected. Additionally, the query can be further specified with a WHERE statement to extract only the data that fulfils a specific condition, such as finding employees whose name is John. The query for this example will be:

```
SELECT name, last_name
FROM employees
WHERE name IS LIKE "John"
```

If at least one employee is named John, a result table will be created with all the outcomes.

By using the statement INSERT INTO followed by the table's name, data can be added to that table. Then the values that need to be added are listed after the VALUES statement.

```
INSERT INTO employees (name, lastname, address, city, country, companyID)
VALUES ("Sarah", "White", "Middle street", "Chicago",  "USA", 8743)
```

2.2.3 Graph databases

Instead of storing information in tables, graph database structures use nodes and edges. The data is stored without a pre-defined structure, allowing flexibility. Graph databases consist of

entities and relationships. The nodes are the entities; labels can specify their specific role, for example, Person. A node can have multiple properties that describe the node. Nodes with the same label do not need the same properties; some nodes can have more properties than others. The edges are relationships and are directed, meaning that they start in one node and are directed to another node. Furthermore, they are provided with an action that explains the relationship; for example, an employee "works" at a company. The relationships in a graph database are the priority and easily extracted, which helps manage deeply linked data (Yoon, S.-K. Kim and S.-Y. Kim, 2017). Angles and Gutierrez state that graph database models are mainly used when information about data interconnectivity or topology is more important or as important as the data itself (Angles and Gutierrez, 2008).

Neo4j

One of the most popular graph database management systems is Neo4j. Neo4j offers a range of applications and tools to store and analyse data. Aura is used to store the data online; therefore, it is always ready to be used. Neo4j graph data science platform helps understand the data using machine learning, resulting in improved predictions and answering critical questions (Neo4j, 2022). They also provide multiple tools for the visualisation of the nodes and edges. Neo4j Bloom lets the user interact with the nodes, and NeoDash lets the user create dashboards based on the data. Using the graph query language Cypher, nodes and relationships can be added to the graph and extract certain relationships between nodes. In an ER diagram, the entities will be the nodes, the relationships are the edges, and each entity's attributes are the nodes' attributes (Jordan, 2014).

Graph Query Language

Adding and extracting data from a graph query is different from a relational database. There is no widely used query language for graph databases, as there is SQL for relational databases. Instead, some languages are developed based on database management systems. A few popular graph query languages are

- Gremlin is a graph programming language from Apache TinkerPop, a graph computing framework for graph databases (OLTP) and graph analytic systems (OLAP) (TinkerPop, 2022). Gremlin is composed of a sequence of steps; each step performs an atomic operation on the data stream. These steps can transform objects in the stream, remove objects from the stream, or compute statistics about the stream.
- SPARQL is a query language developed by the World Wide Web Consortium (W3C), an organisation that designed the standards for the world wide web (W3C, 2013. SPARQL is designed for Resource Description Framework (RDF) databases, also created by W3C, and enables users to query information from databases or any data source stored in RDF format.

Cypher

Another graph query language is Cypher, developed by Neo4j. Cypher is a declarative query language that provides for very efficient reading and writing of data within Neo4j (Jordan, 2014). Cypher resembles the working of SQL in terms of creating nodes. However, for graph databases, relationships play an essential role; relationships must be created using a query. The INSERT statement from SQL is comparable to the CREATE statement of Cypher. With the CREATE statement, nodes can be added to the database.

```
CREATE (n:Company { name : 'CompanyName', description : 'CompanyDescription' })
```

To create a relationship between two nodes, the MATCH statement is used as follows:

```
MATCH (a {employeeName:John}), (b {companyID:8743}) MERGE (a)-[:WORKS_AT
{startDate:'01-01-2014'}]->(b)
```

Besides creating nodes with the MATCH statement, it can be seen as the SELECT statement in SQL and is used to extract nodes or edges that match a specific condition. The following example query returns all the employees that work at the company CompanyName:

```
MATCH employee-[:WORKS_AT]->(b:Company { name: "CompanyName"}) RETURN employee
```

2.3 Graph Theory

2.3.1 Introduction to graph theory

Graph theory is the study of graphs. A graph in mathematics is a pair G = (V, E) of sets. Where elements of V are the vertices, the nodes of the graph G and E are the edges or relationships (Diestel, 2017). In visualising such a graph, the vertices are dots joined by each other through lines if the two corresponding vertices form an edge. If an edge connects two vertices x and y of G, they are called adjacent; otherwise, they are called disjoint. Adjacent vertices can be seen as neighbours (Voloshin, 2009). In graph theory, the degree of a vertex is the number of edges connected to it; this is equal to the number of neighbours. A graph in which every vertex has the same degree is called regular. An example graph can be seen in figures 2.1 to 2.3, created with the online tool https://graphonline.ru/en/about. In figure 2.1, vertices 1 and 2 are adjacent, but vertices 1 and 4 are disjoint. The degree of node 1 is two since it has two neighbouring nodes.

In a directed graph, the edges link two vertices asymmetrically; in an undirected graph,

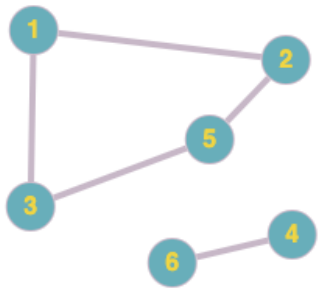

Figure 2.1: Undirected graph

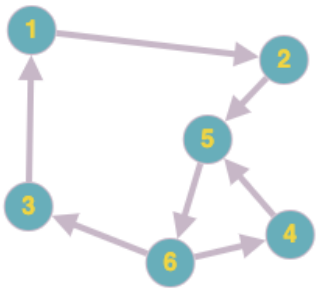

Figure 2.2: Directed connected graph

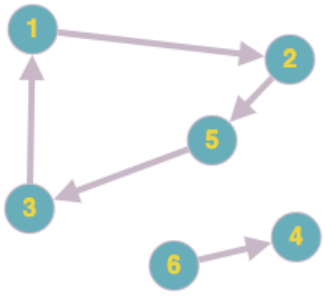

Figure 2.3: Directed disconnected graph

the link between two vertices is symmetrical. Graphs in which order is not important are called undirected graphs, and when they also do not consist of loops but have multiple edges, they are called simple graphs. (Voloshin, 2009). Figure 2.1 visualises an undirected graph, and Figure 2.2 is a directed graph. A graph in which every pair of vertices is an edge is called complete because a new edge cannot be added and obtain a simple graph (Benjamin, Chartrand and Zhang, 2015).

A path is when all vertices can be numbered so that there is precisely one edge connecting every two consecutive vertices and there are no other edges (Benjamin, Chartrand and Zhang, 2015). The path's length is the number of edges encountered when 'walking' through them. For example, in figure 2.2, the paths from nodes 1 to 3 are 1-2-5-6-4-5-6-3 and 1-2-5-6-3. The first path has a length of 7 and the latter 4. When some path connects any two vertices in a graph, the graph is called connected; otherwise, the graph is disconnected (Diestel, 2017). A pair of vertices in a disconnected graph always has no path connecting them. Figure 2.2 represents a connected graph, and figure 2.3 is a disconnected graph.

2.3.2 Measurements

An important question where graph theory offers an understanding is how the value of one invariant influences that of another (Diestel, 2017). Graph theory and network analysis are separate fields; however, they complement each other when analysing nodes/vertices. A graph is the more mathematically version, consisting of vertices and edges. In contrast, the term network is more about graphs representing real-world objects, consisting of entities and relationships between them (Estrada, 2015). Network analysis is occupied with analysing nodes within a network and comparing networks based on network-level metrics.

Node centrality

In the analysis of social networks, node centrality is heavily studied to capture the importance of nodes; however, this metric is also relevant for any other kind of networked system (Estrada, 2015). A few different metrics measure node centrality, each having a different purpose in the analysis.

Degree centrality As previously mentioned, the degree of a node is the number of neighbouring nodes it has. The degree centrality of a node is, therefore, its degree. In a directed graph, a distinction is made between in-degree and out-degree, meaning the number of edges going in the node and outwards. The higher the degree of a node, the more central it is (Nooy, Mrvar and Batagelj, 2005).

Closeness centrality Another centrality metric is closeness, based on the total distance between one node and all other nodes (Nooy, Mrvar and Batagelj, 2005). A node with the highest closeness centrality is the one that is closest to all other nodes in the network. A node with a high closeness centrality can spread information, for example, efficiently through the network since the distances to other nodes are relatively small.

Betweenness centrality Betweenness centrality refers to how often a node is a bridge between other nodes, quantifying the importance of communication between other pairs of nodes in the network (Estrada, 2015). The betweenness centrality of a node is the number of times a node lies on the shortest path between two other nodes. If a node does not use short paths among pairs of nodes, information flows slower through the network (Pinheiro, 2011). Therefore, nodes with high betweenness centrality are often important controllers of power or information.

Eigenvector centrality It can happen in a network where a specific node's closeness centrality is not high but has neighbours whose closeness centrality is high. The eigenvector centrality considers how central a node is by checking how central its neighbouring nodes are. For calculating the eigenvector centrality of a node, the weighted sum of the centralities of neighbouring

nodes is taken. A greater eigenvector might represent more nodes that can be reached with the same effort (Pinheiro, 2011).

Chapter 3

Data

3.1 Exploration results

The data used for this project was taken from the Kumu map, created before starting the project (Zari and Hecht, 2020b) with a few added entries. The data consists of 112 buildings and 145 design strategies. However, 104 buildings were left in the dataset after the removal of buildings having no implemented design strategies. Each design strategy can be linked to one or more of the 59 ecosystem services present in the dataset. These ecosystem services can then be categorised into three groups: cultural services, provisioning services, and regulating services. The data was presented in an excel file with multiple sheets containing the different data entries. Buildings containing information about their location and implemented design strategies are in one table, the specific design strategies linked to an ecosystem service are in a table, and the ecosystem services linked to an ecosystem group are in another table.

Information about the buildings implementing sustainable design strategies was retrieved from the buildings' websites.

3.2 Data preparation

Some entries were missing from the provided dataset. The missing data ranged from a building having no design strategy to a building not having a website link or image. The buildings that did not have a design strategy were deleted. Buildings missing only one or two less essential attributes were kept in the dataset since this would not negatively affect the tool's creation or data analysis.

Some design strategies linked to buildings were not unique since one design strategy could have different names. These terms referring to one design strategy were not removed since this required specific knowledge about ecosystems and design strategies.

3.3 Ethical and legal considerations

Data about the buildings was taken from the buildings' websites or articles online; therefore, all the data is public and does not consist of personal data.

Chapter 4
Methods

This project aims to create an interactive and adaptive tool that visualises the buildings and their implemented design strategies in the dataset. The backend and frontend of this tool will be described separately in this section. The backend is responsible for storing the data and making it adaptive to change, for example, when buildings are added to the dataset or modified by adding new design strategies. The frontend is a website with interactive elements, allowing the user to interact with the data.

4.1 Selection of methods for analysis

4.1.1 Backend

Based on conversations with the supervisor, the initial data structure was adjusted into a clear structure, making adding buildings and design strategies simple. The tool gets its input from this data; therefore, it was needed to implement a straightforward data design. This design consists of three tables, one with all the buildings and their information, one with the design strategies and their ecosystem services, and the last consisting of the ecosystem services and the ecosystem group to which they belong. Figure 4.1 visualises the structure of the data as an ER model.

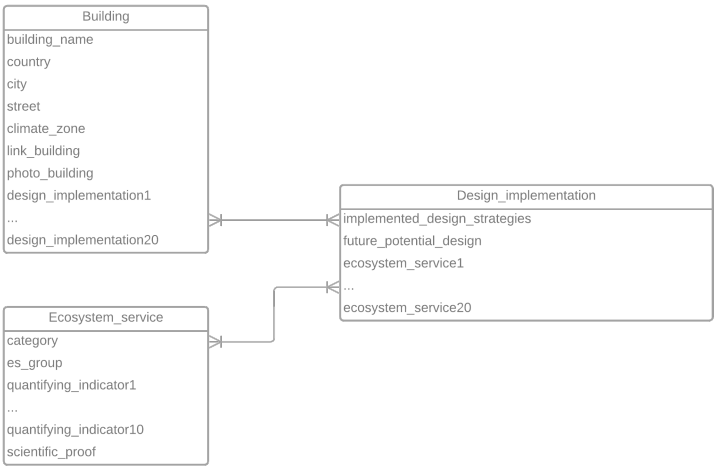

Figure 4.1: Entity Relationship model of the data

Since buildings can implement more than one design strategy and links between multiple ecosystem services exist, a graph database structure was chosen to store the data since it allows for interlinked relationships between entities. Since the tool needs to be adaptive and prone to any future change made to the data, the data is stored in Google Sheets. The user can update the data by adding new rows if new buildings need to be added or when new design strategies are used in buildings. Google Sheets was considered the best option for the initial data storage since it is an online environment. People can be added as editors to the worksheet, making it a collaborative tool where multiple people can modify the data. However, the Google Sheets is set to private; therefore, only people with whom the file is shared can edit the data.

Google Sheets connection

Extracting the data from Google Sheets was done by a python script using Google's library gspread. To make the connection with Google in combination with this library, a project needed to be created on the Google Cloud Platform. A JSON file containing the credentials, such as the authentication information and client email was downloaded from the platform and added to the Python script map structure. The Google Sheets file was shared with the client email address to establish the connection between the Python code and the data inside the worksheet. The data from the Google Sheets was then added to a Pandas data frame using Python, allowing modifying the data through code. Appendix A.1 shows the code used for the Google Sheets connection and the function used to get the records in the worksheet and add them to the data frame.

Neo4j connection

As mentioned before, a graph database structure was used for data storage. A graph database was chosen over a relational database since the focus is on the relationships between buildings

and design strategies, and a building can have implemented more than one design strategy. Furthermore, the Kumu map visualises the data in a graph with multiple layers and having the data in a network allows for a better understanding of the data. Therefore, it seemed logical to stay with visualisation in the form of a network with nodes. Neo4j was chosen since it is a popular choice when storing data in a graph database, and they provide drivers that allow making a connection with the Neo4j environment using Python. Furthermore, retrieving data by Cypher queries is fast using Neo4j, even with large datasets, and the creation of relationships between nodes can be simply added without having to model it first. Using Neo4j's library and Cypher queries, nodes and relationships between nodes could be created. Buildings, design strategies, and ecosystem services were added as nodes by a function written in Python that takes several properties from the node and adds them into a general Cypher query for adding nodes (Appendix A.3). The relationships between the nodes were added similarly, with a different function containing a different Cypher query to add relationships.

Adaptivity

The above connections were merged into one Python script handling the connection to Google Sheets and Neo4j. After connecting to Google Sheets, a check is performed to check if the sheet contains data. If so, the pre-existing nodes and relationships in Neo4j are deleted to empty the database. If the sheet does not contain data, nothing will happen to update the data. When the sheet contains data, and the Neo4j database is empty, the nodes and relationships are created and sent to the Neo4j database.

This script was converted into an executable file using Pyinstaller and then added to a task scheduler to run this program at any desirable moment. This method allows for adaptivity of the data since the database will be relatively up-to-date, depending on how often the executable file runs the Python code.

4.1.2 Frontend

For the development of the tool, a website is created that visualises the data stored in the database and lets users interact with it.

Flask

Flask is used to create the website since it can implement Python code, and it allows the developer to choose what Python libraries and databases the app will use. Figure 4.2 visualises the folder structure for the app. All libraries were imported in the ___init___.py file, and this file is also responsible for launching the app and the routing inside the website. For each route, a function is written with specific code for the page on the website and the redirection to the corresponding HTML file. These HTML files are saved in the templates folder. All HTML pages follow the

same structure set in the base.html file. By using blocks in the base.html file, python code, HTML or Javascript code could be added to structure the layout of the webpages according to what is displayed. Blocks are part of Flask's syntax and allow the developer to insert a piece of code at a specific part of the webpage. For the layout and styling of the website, Bootstrap is used since it is a free and open-source framework for the development of responsive websites.

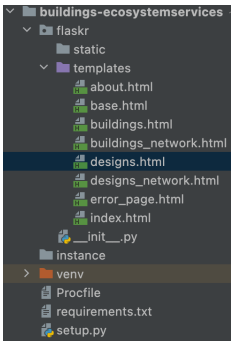

Figure 4.2: Folder structure of the application

Creating data tables with Dash For the creation of interactive tables to showcase the data about buildings and design strategies, Plotly's Dash was used. Dash is an open-source framework for building data visualisation interfaces built on Python. With these data tables, users can filter the data by typing in a letter or word or sorting them alphabetically.

Creating a network visualisation using Neovis Neovis.js is used to create visualisations of nodes and their relationship in a network. This library uses the JavaScript Neo4j driver to connect to and fetch data from Neo4j and a JavaScript library for visualisation called vis.js for rendering graph visualisation. The code to draw the visualisation to the screen was added to the buildings_network.html and the designs_network.html file. In the code, the developer can provide a Cypher query to each instance of the visualisation element; this will return a specific graph drawn to the screen.

Getting network measurements using Networkx To get the graph theory measurements for nodes in the network, the Python library Networkx was used. First, all output nodes were added to a Python list based on a Cypher query. Then, based on this list, a graph was created with Networkx. Finally, with this graph, several measurements could be established that could be linked to specific nodes in the network. The top 10 best-performing nodes for each measurement were selected and outputted on the website under the buildings and design implementations tab.

Publishing the Flask web application using Heroku

The Flask application was published to the web by using Heroku. Heroku is a platform as a service that enables developers to build, run, and operate applications entirely in the cloud.

Heroku was chosen since it is a standard solution for publishing Flask applications to the web and is free. Heroku takes the code from a GitHub repository. Therefore, the folder containing the Flask app is added to Github. This repository was also cloned using git to have a local version of the code, allowing modifying the code and easily pushing the adjusted code to GitHub. A Heroku app was created to connect the Flask project to Heroku; then, the code is deployed to Heroku using the command 'git push heroku main'.

For Heroku to know what happens when users enter the website, the procfile is added. It consists of a line of code that tells which file must be opened to get the application running. Gunicorn is used to serve the Flask application at Heroku; this is a Python Web Server Gateway Interface (WSGI) HTTP server. A WSGI is an interface between web servers and web application frameworks written in Python, in this case, Flask. In the procfile, flaskr is the directory where the application is in, create_app() represents the function's name to start the application.

```
web: gunicorn "flaskr:create_app()"
```

Furthermore, a file containing the requirements is added to ensure the website will work and for Heroku to detect it as a Python project. In the requirements.txt, the version of all the libraries is stated, for example, that of Flask and Dash. In the setup.py file, the setuptools library is called to distribute Python libraries and extensions.

Providing security

The Flask application uses credentials to connect to the Neo4j database for extracting data. However, these credentials have to be a secret to the user of the tool. Otherwise, they can log in to the database and adjust the data themselves. To prevent this from happening, credentials are stored in environment variables, so they are not visible in the applications' source code. Heroku lets developers store their credentials online, which are then easily retrieved in the source code by the operating system (OS) module. By calling os.environ, the user's environmental variables are returned, and outsiders will not see the credentials.

```
import os

url = os.environ['NEO4j_URL']
username = os.environ['NEO4j_USERNAME']
password = os.environ['NEO4j_PASSWORD']
```

Chapter 5
Results

5.1 The final tool

The tool created with this project is a website where users can explore sustainable buildings and design strategies. Based on the Kumu map created before this project, this website tries to visualise the data clearer and cleaner. The requirements for the tool were that it had to be both adaptive to change and interactive. The complete directory for creating the website can be found here: https://github.com/ErtnerV/buildings-ecosystemservices, and the link to the website: https://buildings-ecosystemservices.herokuapp.com/.

5.1.1 Adaptivity

The data in the database is relatively adaptive to change since a program can be run automatically to the desired amount of times a day or week. However, this means that the data is not updated in real-time. Once changes have been made in the Google Sheet, it will only be visible on the website after the executable file is run. This level of adaptivity is sufficient since the data in the dataset is not used in situations where real-time data is necessary. The website will still be helpful when it is not entirely up to date.

5.1.2 Interactivity

The user can click the buttons attached to specific sections on the website's homepage to be forwarded to that page. Furthermore, the user can navigate to different website sections by using the navigation bar on top. Each page has a particular goal, and the navigation bar text clarifies what each page will entail.

The user can select a building from the dropdown menu on the 'buildings network' page. Once a building is set, two graphs will become visible. The graph on the top displays the building as a node connected to all its implemented design strategies as nodes. The graph below visualises the selected building and other buildings implementing the same design strategies. The user can move the nodes to restructure the graph for readability and stabilise them by clicking

on the button to the right of the graph. A similar graph with the same functions will appear on the design network page. The user must select a design strategy from the dropdown menu to see the chosen design strategies and ecosystem services.

The user can browse through all buildings present in the dataset on the building page and see all design strategies on the designs page. This table lets the user sort the data in a column alphabetically by clicking the arrows next to the column headers. Furthermore, the user can filter the data in a specific column by typing in a letter or a word in that column below the column header. The data will be adjusted to the filter once a result can be found in the dataset.

5.1.3 Graph theory

The nodes in the network are analysed using graph theory. Different metrics can be used to calculate the importance of nodes in a graph network. The measurements are calculated based on relatively simple graphs; for example, the degree centrality of buildings is measured based on the graph containing buildings connected to design strategies. For the measurements of the design strategies, a graph is created based on their relationship with ecosystem services. The building with the highest degree and, therefore, having the most connections with design implementations is 'Council House 2'. The design strategy using most ecosystems is 'Medicinal gardens'. For the design strategies, the node with the highest closeness centrality is 'Revegetation', meaning that this strategy is closest to all other buildings in the network. The strategy with the highest eigenvector centrality is 'Revegetation', meaning that based on the centrality of its neighbours, this strategy is the most central.

5.2 Steps in the design process

The design process of the tool followed a few steps, where considerations were made.

5.2.1 Step 1 - Connection to Neo4j and Neo4j dashboard

Since the data in the Kumu map is visualising the data in a circular construction where data is linked, the decision was made to add the data to the graph database Neo4j. Neo4j has a platform named Dash, where developers can create dashboards. A log-in with Neo4j credentials is required for users to access the dashboard. However, the tool needs to be available to everyone; therefore, it was decided that this dashboard is insufficient as a final tool.

5.2.2 Step 2 - Creating a web application to visualise the network

A web application had to be built for the tool to be available on the web. Flask was used to create a web application; however, it was still unclear how to add the graph to the website.

First, a graph was created using Networkx; however, this graph was not interactive. Finally, the graph was made using Neovis.js since the user can move the nodes. At this stage, the graph was showing in the web application, but there was not an option where the user could adjust the Cypher query. Therefore, the cypher query was hardcoded, displaying the same graph every time.

5.2.3 Step 3 - Letting the user adjust the Cypher query and deployment to Heroku

A dropdown menu was added to the network page where users can select a building, allowing interactivity. The output is a graph visualising a network based on the user's decision. For the deployment of the application to the web, a few self-hosted options, as well as hosting platforms are available. The decision was made to use the hosting platform Heroku since they will take care of the maintenance and a domain name; furthermore, it is free to use.

5.2.4 Step 4 - Adjustments to the code for the final tool

The application makes a connection to the Neo4j database using credentials. However, these credentials must be secure and not visible to the users. Heroku's environment variables are used to prevent the credentials from being located in the source code. Furthermore, to visualise the graph network by Neovis, the connection has to be encrypted.

Chapter 6
Discussion

6.1 Main findings

This study aimed to explore if it is possible to create a tool that interactively visualises data. The results show that such a tool can be created with Flask. In combination with Neo4j and several Python libraries, an interactive graph network was created that let users decide the input and interact with the graph.

6.2 Limitations

Some data entries were inconsistent, meaning a design strategy could be referred to by multiple terms. This could give an untrue representation of the actual result since meaningful relationships between buildings, designs, and ecosystem services could be overlooked. Furthermore, potential users have not tested the tool; it is designed based on the developer's taste. Therefore, how users interact with the website and if it is structured logically to gain information remains unknown.

6.3 Further research

The tool can be improved in the future by adding more features that contribute to the mission to inform users about sustainable buildings and design strategies. To improve the current website, potential users must test it. A meaningful addition to the website could be an explanation of the design strategies when a user clicks on a design in the table. Another implementation that improves interactivity would be the option to click a node in the network and receive information about the specific node or create a relationship with another node attached to the specific node.

Finally, to maximise the website's usefulness, one term per design strategy has to be used in the dataset to prevent duplication of strategies.

Chapter 7
Conclusion

This study aimed to develop a tool for users to learn about buildings that implement sustainable design strategies. This tool became a web application created using Flask and data derived from a Neo4j database. The website provides interactivity through a graph network where users can drag nodes and a data table where users can sort and filter data. Based on the results, it can be concluded that interactivity can be implemented into a tool to visualise data.

References

Abdellatif, Mawada and Ahmed Al-Shamma'a (2015). 'Review of sustainability in buildings'. In: *Sustainable Cities and Society* 14, pp. 171–177. ISSN: 2210-6707. DOI: https://doi.org/10.1016/j.scs.2014.09.002. URL: https://www.sciencedirect.com/science/article/pii/S2210670714001000 (cit. on p. 4).

Angles, Renzo and Claudio Gutierrez (Feb. 2008). 'Survey of graph database models'. In: *ACM Computing Surveys* 40 (1), pp. 1–39. ISSN: 0360-0300. DOI: 10.1145/1322432.1322433 (cit. on p. 6).

Benjamin, Arthur, Gary Chartrand and Ping Zhang (2015). *The Fascinating World of Graph Theory*. Princeton University Press. ISBN: 9780691163819. URL: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=852743&site=ehost-live (cit. on p. 8).

Bhatia, Ashima Bhatnagar and Ashima Bhatnagar Bhatia (2014). *Database Management System*. Alpha Science International. ISBN: 9781783321919. URL: http://ebookcentral.proquest.com/lib/uunl/detail.action?docID=5218421 (cit. on p. 4).

Diestel, Reinhard (2017). *Graph Theory*. Vol. 173. Springer Berlin Heidelberg. ISBN: 978-3-662-53621-6. DOI: 10.1007/978-3-662-53622-3 (cit. on pp. 7–9).

Doughty, Mark R C and Geoffrey P Hammond (2004). 'Sustainability and the built environment at and beyond the city scale'. In: *Building and Environment* 39 (10), pp. 1223–1233. ISSN: 0360-1323. DOI: https://doi.org/10.1016/j.buildenv.2004.03.008. URL: https://www.sciencedirect.com/science/article/pii/S0360132304001131 (cit. on p. 3).

Estrada, Ernesto (Mar. 2015). *Graph and Network Theory*. DOI: 10.1002/3527600434.eap726 (cit. on p. 9).

Gordon, Keith (2017). *Modelling Business Information : Entity Relationship and Class Modelling for Business Analysts*. BCS, The Chartered Institute for IT. ISBN: 9781780173535. URL: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=1533376&site=ehost-live (cit. on p. 4).

Grant, Gary (2012). *Ecosystem Services Come to Town : Greening Cities by Working with Nature*. John Wiley Sons, Incorporated. ISBN: 9781118387887. URL: http://ebookcentral.proquest.com/lib/uunl/detail.action?docID=977924 (cit. on pp. 3, 4).

Harrington, Jan L (2003). *SQL Clearly Explained*. Vol. 2nd ed. Morgan Kaufmann. ISBN: 9781558608764. URL: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=209356&site=ehost-live (cit. on p. 5).

– (2009). *Relational Database Design and Implementation : Clearly Explained*. Vol. 3rd ed. Morgan Kaufmann. ISBN: 9780123747303. URL: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=319641&site=ehost-live (cit. on pp. 4, 5).

Jordan, Gregory (2014). *Practical Neo4j*. Apress. ISBN: 978-1-4842-0023-0. DOI: 10.1007/978-1-4842-0022-3 (cit. on pp. 6, 7).

Neo4j (2022). *Neo4j Graph Data Science*. URL: https://neo4j.com/product/graph-data-science/ (visited on 5th June 2022) (cit. on p. 6).

Nooy, Wouter de, Andrej Mrvar and Vladimir Batagelj (2005). *Exploratory Social Network Analysis with Pajek*. Cambridge University Press. ISBN: 9780521841733. URL: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=138973&site=ehost-live (cit. on p. 9).

Norberg, Jon et al. (2012). 'Eco-evolutionary responses of biodiversity to climate change'. In: *Nature Climate Change* 2 (10), pp. 747–751. ISSN: 1758-6798. DOI: 10.1038/nclimate1588. URL: https://doi.org/10.1038/nclimate1588 (cit. on p. 1).

Pinheiro, Carlos Andre Reis (2011). *Social Network Analysis in Telecommunications*. Wiley. ISBN: 9780470647547. URL: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=364870&site=ehost-live (cit. on pp. 9, 10).

TinkerPop, Apache (2022). *Introduction to Gremlin*. URL: https://tinkerpop.apache.org/gremlin.html (visited on 30th May 2022) (cit. on p. 6).

Voloshin, Vitaly I (2009). *Introduction to Graph Theory*. Nova Science Publishers, Incorporated. ISBN: 9781614701132. URL: http://ebookcentral.proquest.com/lib/uunl/detail.action?docID=3019331 (cit. on pp. 7, 8).

W3C (2013). *SPARQL Query Language for RDF*. URL: https://www.w3.org/TR/rdf-sparql-query/ (visited on 30th May 2022) (cit. on p. 6).

Wratten, Stephen et al. (2013). *Ecosystem Services in Agricultural and Urban Landscapes*. John Wiley Sons, Incorporated. ISBN: 9781118506240. URL: http://ebookcentral.proquest.com/lib/uunl/detail.action?docID=1120746 (cit. on p. 3).

Yoon, Byoung-Ha, Seon-Kyu Kim and Seon-Young Kim (2017). 'Use of Graph Database for the Integration of Heterogeneous Biological Data'. In: *Genomics Informatics* 15 (1), p. 19. ISSN: 2234-0742. DOI: 10.5808/GI.2017.15.1.19 (cit. on p. 6).

Zari, Maibritt Pedersen (Jan. 2015). 'Ecosystem Services Analysis in Response to Biodiversity Loss Caused by the Built Environment'. In: *S.A.P.I.E.N.S* 7, pp. 1–14 (cit. on pp. 1, 3).

Zari, Maibritt Pedersen and Katharina Hecht (May 2020a). 'Biomimicry for Regenerative Built Environments: Mapping Design Strategies for Producing Ecosystem Services'. In: *Biomimetics* 5 (2), p. 18. ISSN: 2313-7673. DOI: 10.3390/biomimetics5020018 (cit. on p. 1).

– (2020b). *Strategies for designing urban ecosystem services diagram.* URL: https://embed.kumu.io/0853bdea62a409d6dc7f00a636bc6457#ecosystem-services-in-urban-environments2503193/provisioning-services (cit. on p. 11).

Appendix A
Python code

A.1 Google Sheets connection

```
import gspread
from oauth2client.service_account import ServiceAccountCredentials
import pandas as pd
from neo4j import GraphDatabase
import numpy as np
from py2neo import Graph

scope = ["https://spreadsheets.google.com/feeds",
"https://www.googleapis.com/auth/spreadsheets",
"https://www.googleapis.com/auth/drive.file",
"https://www.googleapis.com/auth/drive"]
creds = ServiceAccountCredentials.from_json_keyfile_name("creds.json",
    scope)
client = gspread.authorize(creds)


def get_buildings_df():
    sheet_buildings = client.open("Buildings-ecosystemservices dataset")
    .worksheet("TABLE 1_examples and designs")

    # Get all the records in the worksheet
    data_buildings = sheet_buildings.get_all_records()
    df_buildings_ = pd.DataFrame(data_buildings)
    df_buildings_ = df_buildings_[df_buildings_.design_implemented1 != '
    ']

    df_buildings_.insert(0, 'id', df_buildings_.index + 10000)
    df_buildings_.insert(1, 'node_type', "Building")
    df_buildings_ = df_buildings_.replace('\'', '', regex=True)
    return df_buildings_


df_buildings = get_buildings_df()
```

A.2 Neo4j connection

```
# Check if there are nodes in the database, then delete them
graph = Graph(url, auth=(username, password))
query = "MATCH (n) RETURN COUNT(n)"
result = graph.query(query)
if result.data()[0]["COUNT(n)"] != 0:
    graph.delete_all()
```

A.3 Create nodes

```
# Create nodes and send them to NEO4j
def creating_nodes(df):
    item_list = df.values.tolist()
    length = len(df.columns)
    data_base_connection = GraphDatabase.driver(uri=url, auth=(username,
    password))
    session = data_base_connection.session()

    for item in item_list:
        create_statement_list = ["CREATE (n" + str(item[0]) + ":" + str(
   item[1]) + " {id:" + str(item[0])]

        for counter, column in enumerate(range(2, length)):
            create_statement_list.append(", " + str(df.columns[column])
   + ":" + "\'" + str(item[2 + counter]) + "\'")

        create_statement_list.append("})")
        final_create_statement = ''.join(create_statement_list)
        session.run(final_create_statement)

creating_nodes(df_buildings)
```

A.4 Create relationship list

```
# Create list with node to node directions - Buildings to designs
design_strategies_dict = pd.Series(df_designs.implemented_design_strategies.values, index=df_designs.id).to_dict()
relationship_list = []
for design_strategy_id, design_strategy_name in design_strategies_dict.items():
    ids = np.unique(df_buildings.stack()[df_buildings.astype('str').stack().str.contains(design_strategy_name)].index.get_level_values(0))

    building_ids = df_buildings["id"].reindex(ids).tolist()
    if building_ids:
        for building_id in building_ids:
            relationship_dict = {
                "from": building_id,
                "to": design_strategy_id,
                "relationship_type": "IMPLEMENTS"
            }
            relationship_list.append(relationship_dict)

df_relations = pd.DataFrame(relationship_list)
rel_list = df_relations.values.tolist()
```

A.5 Executing relationships to Neo4j

```
rel_execution_commands = []
for j in rel_list:
    neo4j_create_statement = "MATCH (a {id:" + str(j[0]) + "}), (b {id:" + str(j[1]) + "}) MERGE (a)-[:" + str(
        j[2]) + "]->(b);"
    rel_execution_commands.append(neo4j_create_statement)


# Create relationships and execute to Neo4j
def execute_relationships(rel_execution_commands_):
    data_base_connection = GraphDatabase.driver(uri=url, auth=(username, password))
    session = data_base_connection.session()
    for rel in rel_execution_commands_:
        session.run(rel)

execute_relationships(rel_execution_commands)
```