



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

*Distributed
Computing*



Wikipedia Walker

Bachelor's Thesis

Niklas Pohl

`nipohl@ethz.ch`

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:

Luca Lanzendörfer, Judy Beestermöller
Prof. Dr. Roger Wattenhofer

August 26, 2023

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisors, Luca Lanzendörfer and Judy Beestermöller, for their invaluable guidance and support throughout this thesis.

I would also like to extend my thanks to Prof. Dr. Roger Wattenhofer and the Distributed Computing Group for providing me with the opportunity and technical support to complete this project

Abstract

This thesis introduces a program designed to master the Wikipedia Game, which is a competitive challenge where participants aim to find the shortest path between two Wikipedia articles using only internal hyperlinks. The program computes real-time paths within the live Wikipedia environment, driven by an up-to-date database of articles and their connections. This database can also be used in the future for further investigations of articles and their respective interconnections. In contrast to existing solutions, this program addresses limitations in path accuracy and link visibility, resulting in a tailored and effective tool for mastering the Wikipedia Game.

Contents

Acknowledgements	i
Abstract	ii
1 Introduction	1
1.1 Wikipedia Game	1
1.2 Related Work	1
2 Collecting Data	3
2.1 Wikipedia API	3
2.2 Challenges	4
2.3 Scraping	5
2.3.1 Initial Scrape	5
2.3.2 Live Feed	6
2.3.3 Monthly Metadata Scraping	6
2.3.4 Redirect Edge Management	7
2.4 Recovery	7
3 Databases	9
3.1 Neo4j	9
3.1.1 Neo4j on Disk	9
3.1.2 Performance	10
3.1.3 Shortest Path	11
3.1.4 Data Layout	12
3.2 Redis	12
4 API	14
4.1 Endpoints	14
4.1.1 Shortest Paths	14

CONTENTS	iv
4.1.2 Interesting Paths	15
4.1.3 Neighborhood	16
4.1.4 Random Walk	17
4.1.5 Search	19
5 Frontend	20
5.1 Framework	20
5.2 Website	20
6 Deploying	24
6.1 Docker	24
Bibliography	26

Introduction

The objective of this thesis is to develop a program capable of mastering the Wikipedia Game. The program's core functionality involves real-time path computation within the live Wikipedia environment, necessitating a consistently up-to-date database of Wikipedia articles and their corresponding connections. This database can also be leveraged for future investigations into articles and their interconnections, serving as a foundation for further exploration.

1.1 Wikipedia Game

The Wikipedia Game is a popular online challenge where participants have to navigate from one Wikipedia article to another using the fewest number of links. The goal of the game is to find the shortest path, in terms of linked articles, between two given articles on Wikipedia. Participants start from a source article and must reach a target article by clicking on hyperlinks within the articles. The catch is that players can only navigate by following the internal links present within the articles; they are not allowed to use the search function or directly input URLs.

1.2 Related Work

There already exists a similar program to ours, called *6 Degrees of Wikipedia* [1] which is able to compute the shortest paths between two Wikipedia articles. However, this program is not well-suited for the Wikipedia game, where players have to navigate from one topic to another as fast as possible, using the fewest number of link clicks due to two main reasons. Firstly, it lacks the ability to navigate through Wikipedia articles effectively, as it only identifies the existence of a connection without providing the actual link names. Secondly, *6 Degrees of Wikipedia* builds its database from Wikipedia dumps, which are updated twice a month. Consequently, it cannot guarantee the shortest path between articles, nor can it ensure the path's existence, given Wikipedia's rapidly changing nature

with numerous edits occurring every hour.

To address these limitations and offer the flexibility to filter the shortest paths based on interesting criteria, such as the most viewed path among all possible shortest paths, we made the decision to develop our own program. This custom-built solution allows us to effectively handle these challenges, providing a more tailored and accurate experience for the Wikipedia game.

Collecting Data

In order to query the data, we must first collect it. Previous programs, such as *6 degrees of Wikipedia* built their graph from a Wikipedia dump, which is provided by Wikipedia twice a month. Initially, we also considered constructing our graph from these dumps. However, during our research, we discovered that not all links are included in these dumps.

As a result, we reached the conclusion that we need to scrape Wikipedia directly from scratch. Scraping Wikipedia allows us to gather the most up-to-date data, ensuring the accuracy and completeness of our graph. By implementing a scraping mechanism, we can extract relevant information directly from Wikipedia's API, including article, links and metadata.

In the following sections, we will dive into the details of our Wikipedia scraping process, exploring the methods employed and the data collected.

2.1 Wikipedia API

Wikipedia provides an API which significantly facilitates the scraping process. The Wikipedia API allows us to filter and retrieve only the relevant data, optimizing the data collection process. Moreover, the API enables us to access additional metadata, such as page views, which would not be accessible through a direct web scrape of Wikipedia.

We decided to collect the following key properties for each Wikipedia article to build our database:

- **Links to Other Wikipedia Pages:** This data is vital for constructing the graph, as it establishes the connections between different articles.
- **Protection Status of the Page:** The protection status of Wikipedia articles indicates the level of editing and moving rights granted to user groups, serving as a safeguard against vandalism and unauthorized changes.

- **Page Views:** The monthly views of the article .
- **Article Length:** The length of the article in bytes.
- **Plain Text:** Instead of storing the HTML code, we opted to save memory space by storing the plain text of the Wikipedia articles which can be directly fetched from the API.

2.2 Challenges

Throughout the scraping process, we encountered several challenges namely:

Lack of link names: In *6 degrees of Wikipedia*, they provided the shortest paths between articles; however, the absence of link names has made it impractical for the Wikipedia game. Recognizing this limitation, we aimed to address it. The issue arose from the inability to directly request actual link names through the Wikipedia API. As a solution, we had to retrieve also the HTML code of the Wikipedia pages and develop a parsing mechanism to extract the precise names of the links. So that they can be found through CTRL + F.

Hidden Links: Furthermore, we encountered links that were present on the page but hidden within drop-down menus, usually located at the end of the page. These hidden links posed a unique challenge, as they could not be discovered merely by searching for the link names, which again made them impractical for our purpose. To address this, we modified our parsing code to detect whether a link was hidden within a drop down menu or if it is visible. This modification enabled us to include an attribute with which we can later filter the paths.

Redirect pages: These are pages containing only one link to another Wikipedia article, and they automatically redirect you to this article. According to the Wikipedia Foundation, the purpose of these redirect pages is to ensure that alternative article names or closely related topics redirect to the most appropriate article, enhancing the ease of searching for specific articles. Redirects are created automatically when an article is moved or manually [2]. The issue we faced with redirects was that they introduced an extra step in our graph representation, counting as a distance of two between two articles Figure 6.1 (left side). However, in practice, users will never land on the redirect page when clicking links, as Wikipedia automatically redirects them to the intended article. To accurately model this behavior in our graph, we decided to mark them as redirect nodes and establish a direct redirect edge between articles without the intermediate redirect page Figure 6.1 (right side).



Figure 2.1: Graph Representation with Internal Redirects: Orange nodes denote redirect pages, while blue nodes indicate article pages

2.3 Scraping

We have organized the data collection process into four distinct sub-programs

1. **Initial Scrape:** The first sub-program is responsible for scraping all Wikipedia articles once to construct the graph.
2. **Continuous Updates:** The second sub-program handles the real-time monitoring of Wikipedia's live change feed. And it promptly updates the graph with the changes made to Wikipedia.
3. **Monthly Metadata Scraping:** The third sub-program scrapes monthly new metadata, including page views or protection status for each article.
4. **Redirect Edge Management:** The fourth sub-program focuses on updating and adding redirect edges within the graph. This ensures that redirect pages are appropriately linked

2.3.1 Initial Scrape

During the planning phase of our scraping process, we explored various techniques to ensure the comprehensive collection of all Wikipedia articles. Initially, we considered using a Breadth-First Search (BFS) algorithm to traverse Wikipedia and gather all articles, but a fully connected graph would be required for this approach to work.

To overcome this, our program dynamically downloads the latest file containing all current Wikipedia article names, provided twice a month by Wikipedia. Utilizing this list, we can individually scrape each article, even if the graph is not fully connected.

For each article, we request its links to other Wikipedia pages, the plain text, and the HTML code to extract link names and check their visibility. We perform this process in parallel to accelerate the data collection, although the number of threads is limited by the Wikipedia API, which will slow down individual requests.

Given Wikipedia’s extensive data, with approximately 17 million articles, including 10 million redirect pages and roughly 700 million distinct links, the complete scraping process took 20 days. Despite its relative slowness compared to building the complete graph from the Wikipedia dump, this approach significantly improves the correctness of our graph.

2.3.2 Live Feed

Having successfully built a complete graph of Wikipedia, the next critical step is to devise a method to keep this graph up to date and incorporate all the latest changes happening on Wikipedia. To achieve this, we leverage Wikipedia’s live recent change feed, which provides a real-time listing of all updates made to Wikipedia. By connecting to this feed, we can monitor the changes and promptly update our graph accordingly. The live recent change feed provides information about articles that have been updated, deleted, created, or moved. Based on this information, we perform the following actions

- **Updated Articles:** If a page is updated, and we have not yet scraped it, we simply ignore the update for the time being. We know that we will eventually collect the newest version of the page during the scraping process. However, if the page has already been scraped, we perform a new scrape for that page to obtain the latest content, links, and text. We then compare this new data to the existing data in our database and update it accordingly. This ensures that our graph remains accurate and reflects the latest information available on Wikipedia.
- **Deleted/Moved/Created Articles:** For articles that have been deleted, moved, or created, we directly propagate the updates to our graph database. We do not wait for scrape in these cases to maintain the correctness of the graph. By promptly updating our graph with these changes.

By continuously tracking all changes on Wikipedia and updating our graph accordingly, we can maintain an almost up-to-date representation of the Wikipedia graph at any given time.

2.3.3 Monthly Metadata Scraping

To maintain up-to-date metadata for our Wikipedia articles, we implement a monthly update process. Each month, we collect all article titles present in our database. We then iterate through each title and request the corresponding article’s length, views, and protection status from the Wikipedia API. Once we have updated all metadata for the articles in our database, the program waits for a month before repeating the process.



Figure 2.2: Left: Dotted node represents an article that has not been scraped yet. Orange node is a Redirect node which has been scraped, and redirect edges have been updated accordingly. Right: Dotted node has been scraped now. However, no redirect edge is present for this article.

2.3.4 Redirect Edge Management

To ensure the integrity of redirect edges within our graph database, we have developed a separate sub-program dedicated to their maintenance. This sub-program regularly iterates through all redirect nodes in our database and examines the incoming edges from articles that link to these redirects. It creates redirect edges for those articles if they do not already exist, pointing them to the appropriate target articles.

There is a reasons why we handle redirect edge maintenance separately from our normal update method. Our normal update method is used for both the initial scrape of nodes and the continuous updates. If we were to add redirect edges during the initial scrape in the same manner described above, we would not have a guarantee that we have added these redirect edges to all nodes that point to these redirects. This is because there may be nodes that we have not yet scraped, which could potentially point to these redirects figure 2.2. Therefore, we would need to iterate through all neighbors of a article node to check if they point to a redirect and add the redirect edge accordingly. Similarly, during updates, we would need to check all new links and examine if any redirect nodes are among them. Additionally, for any deleted links in the update, we would need to remove potential redirect nodes. Performing these operations within the normal update method would introduce significant overhead and slow down the overall process. Thus, to ensure efficient updating and keep pace with the live feed, we implemented this functionality in a separate sub-program.

2.4 Recovery

To address the challenge of the initial scrape's relative duration and the inconvenience of rebuilding the entire graph from scratch after server maintenance, we have implemented a recovery mode that ensures the up-to-date version of the

graph, even if the program was down for a few days.

Therefore, we store the `rcid` (revision change ID) in our database, which is a unique number provided by the Wikipedia API during the live feed scraping. This `rcid` indicates the latest update received, along with the date of the call. When the program restarts, it checks for the presence of this `rcid` in the database and verifies if the server downtime was within a predefined threshold period. If this condition is met, the program retrieves the latest update from the feed and updates the nodes accordingly until it is up-to-date again. However, if the downtime exceeds the predefined threshold, we need to clear the databases and initiate the initial scrape.

Databases

In the initial stages of the thesis, we conducted a comparison between SQL Databases and Graph Databases to determine the most suitable option for our requirements. Through a series of small-scale tests and by referring to the paper titled *The Shortest Path Algorithm Performance Comparison In Graph and Relational Database on a Transportation network* [3], which evaluates the performance of Postgres, an SQL database system, and Neo4j, a Graph Database, in finding the shortest path. The findings of the paper indicated that Neo4j achieved up to a 35% better performance compared to the Postgres database, despite its higher memory consumption. Given our primary concern for performance rather than memory usage, we opted to utilize Neo4j as our primary database.

3.1 Neo4j

Neo4j [4] is a graph database management system that offers a different approach to storing and querying data as traditional relational databases. Neo4j does not rely on tables and rows, instead it leverages the graph data model to represent data.

In Neo4j, nodes represent entities or objects, while relationships define the connections between these entities. Nodes can have properties that store additional information, and relationships can also have properties to capture attributes of the connections. This graph-based representation allows for way more efficient traversal and analysis of complex relationships.

3.1.1 Neo4j on Disk

On disk, Neo4j stores data in linked lists. All properties of a node are stored in a single list, where each element contains the actual property value and a pointer to the next property in the list. Similarly, relationships between nodes are stored in linked lists, with elements containing pointers to the start and end nodes, as well as a pointer to the next relationship between these nodes.

The nodes themselves have two pointers: one pointing to the start of the properties linked list and the other to the start of the relationships linked list. This memory schema allows for extreme flexibility, as each node, even if they belong to the same type, can have a different number of properties [5].

However, this flexibility comes with a drawback in terms of search performance. When searching for a specific property, all nodes have to be queried, and each property list must be examined, which can lead to relatively slow search operations.

3.1.2 Performance

We conducted a series of performance tests to determine the impact of data modeling on achieving optimal performance. These tests were performed on the German Wikipedia dump, which consisted of approximately 5 million nodes and roughly 200 million edges. Our initial investigation focused on evaluating the influence of large nodes, i.e., nodes with a significant amount of data, on the runtime of Neo4j's shortest path algorithm.

To investigate this, we created two separate instances of the graph based on the German Wikipedia dump. In the first instance, the nodes only stored their relationships with other nodes and did not contain any additional data. In the second instance, the nodes were enriched with extensive additional data, including the text from the corresponding Wikipedia articles and additional metadata. We then selected the same subset of nodes in both instances and computed the shortest path between them.

In our tests Neo4j performs approximately 50% better when the nodes do not contain additional data or have less data compared to the nodes with substantial additional data. Furthermore, it can be observed that this performance improvement does not increase as the path length between two nodes grows.

The second test focused on evaluating the performance of the shortest path algorithm when constantly deleting nodes and edges in the graph. Since we continuously scrape Wikipedia and update our database with changes, the database needs to handle such operations, which may occur frequently.

For this experiment, we again selected a subset of nodes and computed the shortest path between a node A and B. Afterward, we deleted a node that was part of the shortest path and reran the algorithm to observe the impact on runtime. The objective was to determine whether Neo4j recomputes the shortest path from scratch.

The result was that the runtime was not affected by the deletion; in fact, it was even faster. However, this improvement in runtime can mainly be attributed to the data already being present in the cache.

In the third and final experiment, we compared the runtime performance of

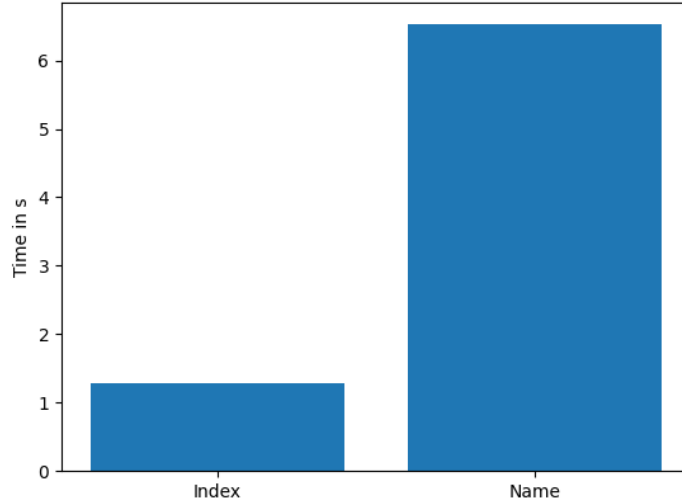


Figure 3.1: The graph displays the average time Neo4j’s shortest path algorithm requires when searching the source and target nodes with their respective article names or directly with their Neo4j index.

Neo4j’s shortest path algorithm when searching for nodes based on their respective Wikipedia article names versus using Neo4j’s index. Each node in the graph had a property called `title`, which represented the name of the article. We conducted the shortest path computation for the same subset of nodes, and the results are presented in Figure 3.1, highlighting the significant difference observed.

Figure 3.1 demonstrates the considerable impact this choice has on the runtime. While Neo4j excels in traversing the graph efficiently from a given node, it struggles when searching for nodes based on specific attributes. This discrepancy can be attributed to the underlying disk storage mechanism employed by Neo4j. On average, we achieved almost an 600% performance boost when utilizing Neo4j’s index instead of searching nodes based on their attributes.

3.1.3 Shortest Path

Internally, Neo4j employs a fast bidirectional breadth-first search algorithm when predicates can be evaluated during traversal [6]. This optimized approach allows for efficient and speedy path computations. However, if the predicates cannot be evaluated during traversal or if we need to consider the entire path as a whole, Neo4j switches to the slower but exhaustive breadth-first search algorithm. This ensures the correctness of the shortest path.

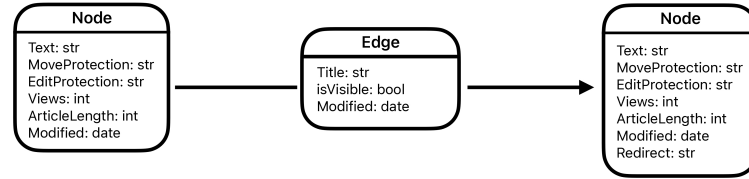


Figure 3.2: Data Layout in the Database where the left node is a normal article node and the right node is a redirect node.

3.1.4 Data Layout

Given that the observed performance downgrades when storing all data in the nodes, while noticeable, do not fall into the category of being excessively extreme, we have chosen to adopt a more convenient approach for data retrieval. This involves directly storing all data in Neo4j nodes, focusing more on indexing, which has demonstrated a significantly higher impact on runtime improvement. Each individual article will be represented as a dedicated node, featuring attributes such as the article's plain text, views over the last month, protection level, article length, modification date, categories, and for redirect nodes, the name of the node to which the redirection points Figure 3.2. Additionally, we will establish directed edges between nodes whenever a link exists between them. These edges will be equipped with attributes such as `isVisible`, indicating whether the link is searchable using `CTRL+F`, `title` representing the link's name, and `date` signifying when the link was added.

3.2 Redis

As our investigation has highlighted, the most substantial efficiency enhancement arises when directly indexing nodes in Neo4j rather than accessing them through properties. Consequently, we require an efficient mapping for titles to indices. This is where Redis enters the equation. Redis, recognized for its remarkable speed as a key-value database, supports a variety of data types, including strings and lists. Notably, operations such as retrieval, setting, or deletion in the database are executed with an $O(1)$ time complexity for string data types [7], making it an optimal choice for fulfilling our title to index mapping.

The reason for our choice of an additional database, as opposed to a simple hash map, lies in the fact that we would otherwise reconstructing the hash map entirely from scratch each time the program restarts would be far from ideal. Now when

searching for nodes based on article names or titles, we can use Redis to quickly retrieve the relevant node id and then directly access those nodes in Neo4j, thus avoiding the slower property-based search.

In our backend, we have chosen to utilize FastAPI [8], which is known for its high performance and efficiency as one of the fastest Python frameworks available. Its lightweight nature and user-friendly design have enabled us to swiftly initiate our project.

Furthermore, to establish direct connections to our databases from Python, we rely on the Python libraries for neo4j and redis.

4.1 Endpoints

4.1.1 Shortest Paths

The `/shortestPaths` endpoint enables users to find the shortest path between two Wikipedia articles. Users can specify the source node and the target node by providing their titles. Additionally, they have the option to set the number of paths to be returned (use `max` to retrieve all possible paths), choose whether to resolve redirects, and filter for visible edges only.

Field	Type	Description
source	string	title of the source node
target	string	title of the target node
numPaths	string	how many paths get returned (max for all)
redirects	bool	resolve redirects
visible	bool	only use visible edges

Implementation

At the beginning of the API call, we check whether the source and target are part of our Redis database. If they exist, we retrieve their respective IDs; otherwise, we return an error to the client. Subsequently, we prepare our Neo4j query for the shortest path, incorporating all the relevant parameters. Firstly, we locate

both the source and target nodes using the received IDs from our previous query. Next, we use the built-in shortest path function from Neo4j to find the shortest paths that meet our requirements, such as whether the algorithm is allowed to use redirect edges or only normal edges, and whether it can use invisible edges. Furthermore, we limit the number of paths returned if the parameter `numPaths` is not set to max. Finally, we return the individual paths with their respective Wikipedia article names, all nodes that are part of a shortest path, and all edges that are part of a shortest path. After retrieving the necessary data from the Neo4j database, we proceed to sort and format the return values. This step is crucial as it enables us to directly input the data into our graph library in the frontend without requiring any additional computation on the client side.

```

1 startID = redis.get(source)
2 endID = redis.get(target)
3
4 ...
5
6 query = f"""
7     MATCH (n) WHERE id(n)={int(startID)}
8     MATCH (m) WHERE id(m)={int(endID)}
9     MATCH p=allShortestPaths((n)-[e{":edge" if not redirects else
10    ""}]*)->(m))
11     {"WHERE all(r IN relationships(p) WHERE r.isVisible)" if
12    visible else ""}
13     WITH nodes(p) AS path, relationships(p) AS rel
14     RETURN [node in path | node.title] as path,
15             [node in path |
16             [id(node),node.title,node.articleLength,node.pageViews
17             ]],
18             [e in rel |
19             [id(startNode(e)),id(endNode(e)),e.title,e.isVisible,
20             type(e)]]
21     {f"LIMIT {int(numberPaths)}" if numberPaths!="max" else ""}
22 """

```

4.1.2 Interesting Paths

The `/interestingPaths` endpoint enables users to filter the shortest path between two Wikipedia articles based on specific metadata from the articles. For instance, users can filter for the most viewed path among all shortest paths. To use this endpoint, users need to specify the source node and the target node by providing their titles. Additionally, they have the option to set the number of paths to be returned (use max to retrieve all possible paths), choose whether to resolve redirects, and filter for visible edges only.

Field	Type	Description
source	string	title of the source node
target	string	title of the target node
numPaths	string	how many paths get returned (max for all)
redirects	bool	resolve redirects
visible	bool	only use visible edges
order	string	ASC (maximize) or DESC (minimize)
attribute	string	the attribute for which we want to filter the shortest paths (e.g., views)

Implementation

The implementation for finding interesting paths is very similar to the one for the shortest path. The key difference is that we now sum up each path for the given attribute and sort the results accordingly, either in ascending or descending order, depending on whether we want to maximize or minimize the attribute value. Apart from this, the rest of the query and function remains almost the same.

```

1 query = f"""
2     ...
3
4         reduce(res=0, x in [node in path|node.{attribute}]]| res + x
5     ) as sum
6     ORDER BY sum {order}
7
8     ...
9 """

```

4.1.3 Neighborhood

The `/neighborhood` endpoint allows users to compute the neighborhood of a given node in the graph. Users can specify the source node by providing its title. Additionally, they have the option to retrieve incoming and outgoing edges, control the number of incoming and outgoing edges to fetch, resolve redirects if needed, and filter for visible edges only.

Field	Type	Description
source	string	title of the node
incoming	bool	if we want to get the incoming edges
numIncoming	string	how many incoming edges ("max" for all)
outgoing	bool	outgoing edges
numOutgoing	string	how many outgoing edges ("max" for all)
redirects	bool	resolve redirects
visible	bool	only want visible edges

Implementation

In the same manner as for the shortest paths, we begin by searching for the node ID of the source article in our Redis database. Next, we query our database for all outgoing edges from this node if the outgoing flag is set to true, limiting the returned neighbors to the specified parameter numOutgoing. Similarly, we do the same for incoming edges. After retrieving all the necessary neighbor nodes from the Neo4j database, we proceed to sort and format the return values, following the same process as we did for the shortest paths.

```

1 nodeID = redis.get(source)
2
3 ...
4
5 query_outgoing = f"""
6     MATCH (n) WHERE id(n)={int(nodeID)}
7     MATCH (n)-[e{":edge" if not redirects else ""}]->(m)
8     {"WHERE e.isVisible" if visible else ""}
9     RETURN
10        [id(startNode(e)),id(endNode(e)),e.title,e.isVisible,type(e)
11        ],
12        [id(m),m.title,m.articleLength,m.pageViews]
13        {f"LIMIT {int(numOutgoing)}" if numOutgoing!="max" else ""}
14    """

```

4.1.4 Random Walk

The `/randomWalk` endpoint allows users to initiate random walks in the graph, starting from a specified source node. Users can determine the length of the random paths `pathLength` and the number of random walks to be performed `numWalks` from the source node. Additionally, users have the option to filter for visible edges only and resolve redirects during the random walk process.

Field	Type	Description
source	string	title of the source node for the random walk
pathLength	int	how long the random paths should be
numWalks	int	how many random walks we take from the source node
visible	bool	only use visible edges
redirect	bool	resolve redirects

Implementation

Since Neo4j does not directly support a random walk function, we have implemented our own. Firstly, we obtain the `nodeID` of our source article. Then, we utilize two nested loops: one for the number of walks and the other for each step in the walk. During each step, we perform two database calls. The first call calculates the number of valid outgoing edges for the current node, and the second randomly selects one edge from them. To achieve this, we query for all outgoing edges and then skip the first $p \times \text{numberEdges}$ edges, where $p \in [0, 1]$ is randomly generated at the beginning of each step. Additionally, we restrict the number of returned nodes to 1, as we require only one random neighbor. We repeat this process until we reach the desired path length. If the path length is reached, we reset the node to the source node and proceed with another walk until the specified number of walks is completed. Afterward, we sort and format the result before returning it to the client.

```

1 nodeID = redis.get(source)
2
3 ...
4
5 for walk in range(numWalks):
6
7     ...
8
9     for length in range(pathLength):
10         p = random.random()
11
12         numEdges = f"""
13             MATCH (n) WHERE id(n)={int(nodeID)}
14             MATCH (n)-[e{":edge" if not redirects else ""}]->(m)
15             {"WHERE all(r IN relationships(p) WHERE r.isVisible)"
16              if visible else ""}
17             RETURN COUNT(*)
18         """
19
20         ...
21
22         source = f"""
23             MATCH (n) WHERE id(n)={int(nodeID)}
24             MATCH (n)-[e{":edge" if not redirects else ""}]->(m)

```

```
25         {"WHERE all(r IN relationships(p) WHERE r.isVisible)"
26         if visible else ""}
27         RETURN
28             COUNT(*) as number,id(m) as id,
29             [id(n),id(m),e.title,e.isVisible,type(e)] as edge,
30             [id(n),n.title] as source,
31             [id(m),m.title] as target
32         SKIP {int(p*numEdges)} LIMIT 1
33     ""
34
35     ...
```

4.1.5 Search

Since our Redis database relies on an exact match, providing the correct spelling of the desired article is crucial, as there is no spelling correction or elastic search. To tackle this challenge, we implemented recommended articles while typing, utilizing the Wikipedia API for searching, which performs the elastic search for us. This ensures that users receive accurate spelling suggestions and recommended articles while searching

Field	Type	Description
id	string	The string which was type until now

Frontend

With the entire infrastructure now in place, encompassing the storage and continuous updates of the Wikipedia graphs, the website serves as a powerful method to showcase and visualize the results.

The primary objective of the website is to present the graphs in an user-friendly manner. By creating an intuitive and interactive platform.

5.1 Framework

For our Frontend, we utilize React [9]. React is a popular and powerful JavaScript library for building interactive and dynamic user interfaces. It is widely adopted in web development due to its component-based architecture, which allows developers to create reusable and modular UI elements. React's virtual DOM efficiently updates only the necessary parts of the UI, leading to enhanced performance and responsiveness. This makes React an ideal choice for building modern, scalable, and maintainable web applications.

Additionally, we have incorporated two essential libraries, namely Redux [10] and react-vis-network-graph [11], to enhance our website's functionality.

The incorporation of Redux makes it easier to manage the state of the application across multiple components in react. This powerful state management library streamlines the handling of data flow and ensures consistency throughout the user interface.

As for rendering the final computed graph to the display, we rely on the react-vis-network-graph library which makes it easy to plot network graphs.

5.2 Website

Our primary objective was to create a user-friendly and aesthetically pleasing website that seamlessly integrates all the required functionalities. We focused on designing a clean and intuitive interface to ensure a smooth and enjoyable user experience while encompassing the full range of functionalities we aimed to offer.

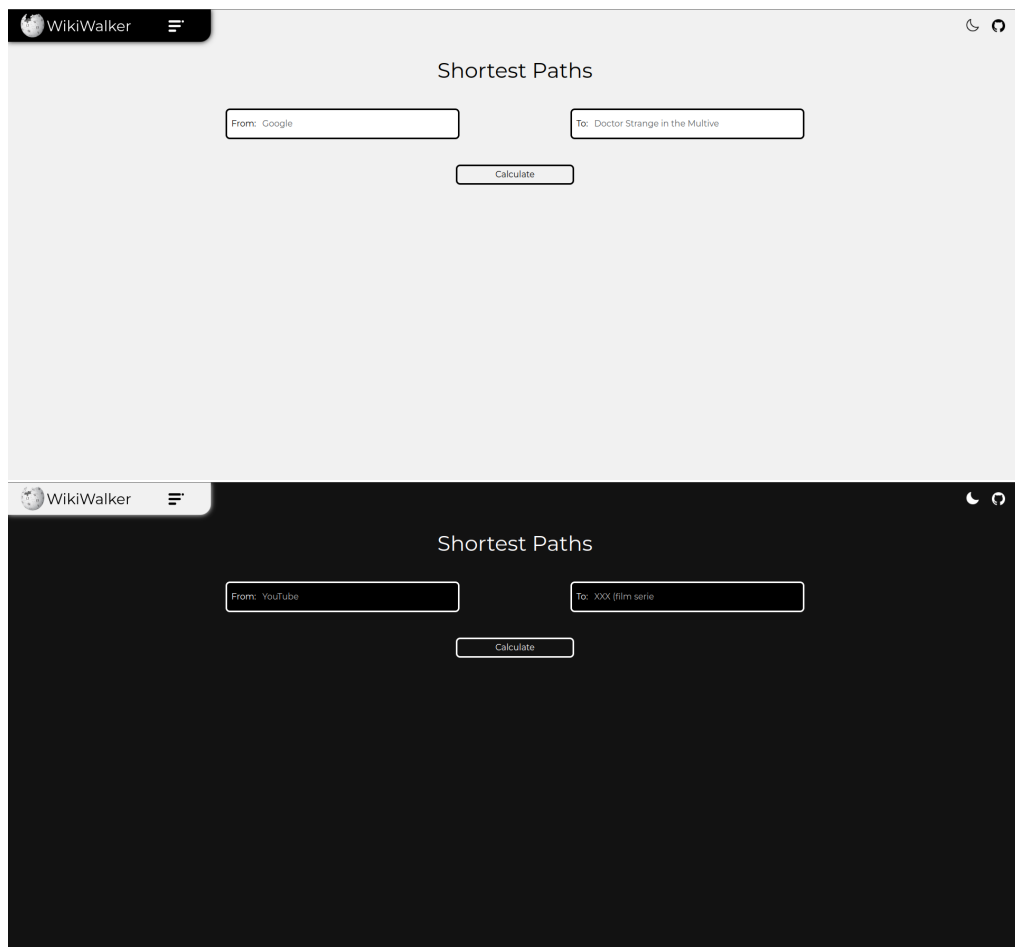


Figure 5.1: The home screen of the website

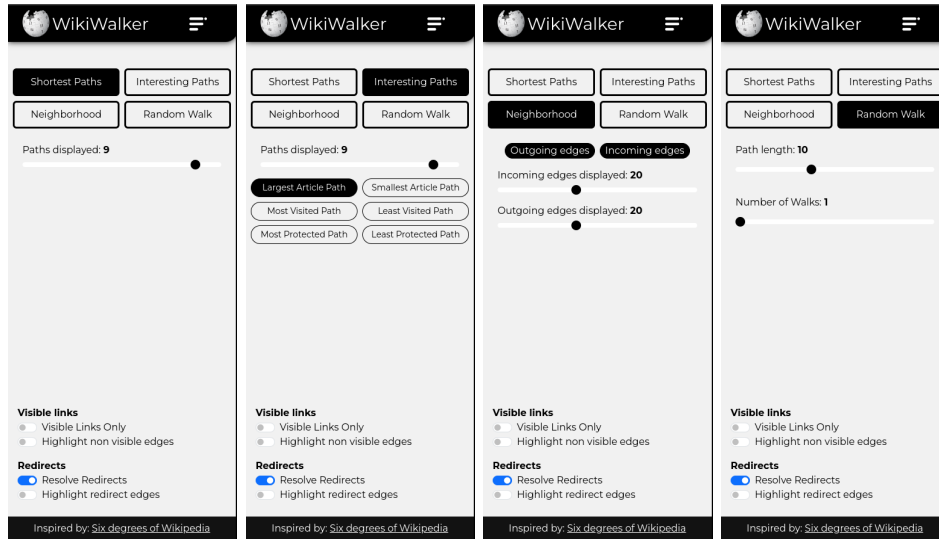


Figure 5.2: The images display all possible features and filters that are available.

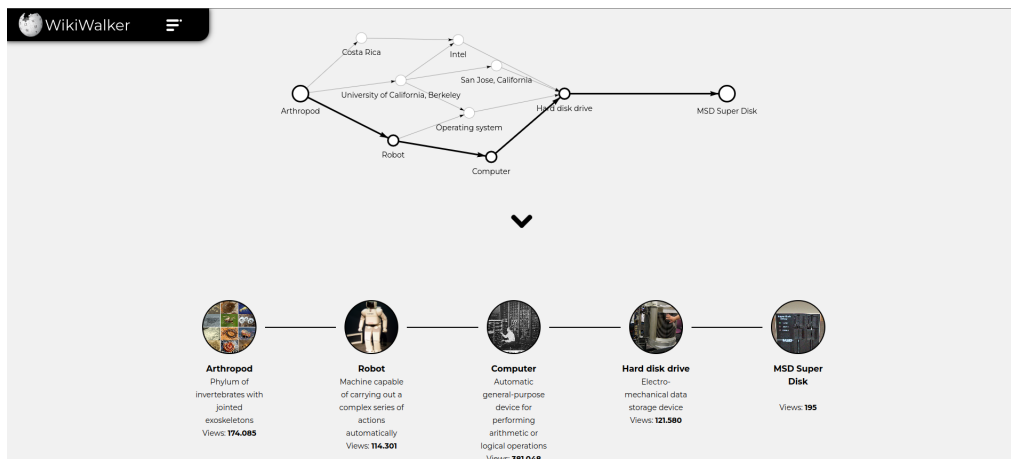


Figure 5.3: Home Screen of the Website Displaying a Graph

Our home screen has a minimalist design, primarily comprising input fields for the desired articles (refer to Figure 5.1). Additional functionalities are tucked away within a collapsible sidebar, providing options to select different modes and tailor the final graph (refer to Figure 5.2). The graph itself maintains a simplicity-first approach, revealing further details upon hovering. Hovering over an edge exposes the actual link name, while hovering over a node provides additional information about the respective article. Positioned below the graph is a display of all paths within it, facilitating individual investigation (refer to Figure 5.3).

Deploying

As a final step, we were required to deploy our program to a server where it can run efficiently. For this purpose, we chose a Linux server with 64 GB RAM and 32 cores, providing the necessary computational power to track all changes on Wikipedia. The initial scrape, using this configuration, took approximately 20 days to complete. The limiting factors were the Wikipedia API, which slowed down our requests if we made too many in a short period, and the memory, as the Neo4j database ultimately occupied 103 GB. Consequently, not the entire database could be loaded into memory, impacting the overall progress. In comparison, the Redis database occupied only 1.5 GB in memory

6.1 Docker

For the deployment of our program, we utilized Docker. Docker serves as a powerful and widely-used containerization platform, simplifying the deployment, management, and execution of applications in an isolated and consistent environment. Containers function independently, encapsulating their own software, libraries, and configurations, while also enabling seamless communication through well-defined channels. Notably, they share the services of a single operating system kernel, leading to resource-efficient performance compared to conventional virtual machines [12].

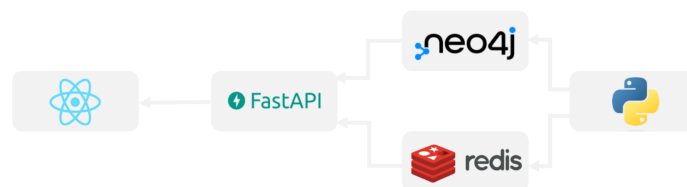


Figure 6.1: The diagram illustrates the program’s architecture, with each block representing a distinct Docker container.

In our case, our program is designed to operate within five distinct Docker containers, each responsible for specific components: one for each database, the API, the website, and the scraping program. Leveraging Docker's portability, our program enjoys the flexibility to run seamlessly on Linux, Windows, or macOS systems, empowering us to effortlessly switch between different environments.

Bibliography

- [1] “6 degree of wikipedia.” Jacob Wenger. [Online]. Available: <https://www.sixdegreesofwikipedia.com/>
- [2] “Redirects.” Wikipedia. [Online]. Available: <https://en.wikipedia.org/wiki/Wikipedia:Redirect>
- [3] D. O. MARIO MILER, DAMIR MEDAK, “The shortest path algorithm performance comparison in graph and relational database on a transportation network.” [Online]. Available: <https://hrcak.srce.hr/file/183346>
- [4] Neo4j. [Online]. Available: <https://neo4j.com>
- [5] “Neo4j on disk.” Neo4j. [Online]. Available: <https://neo4j.com/developer/kb/understanding-data-on-disk/>
- [6] “Shortest path planning.” Neo4j. [Online]. Available: <https://neo4j.com/docs/cypher-manual/current/execution-plans/shortestpath-planning/>
- [7] Redis. [Online]. Available: <https://redis.io/docs/about/>
- [8] Fast API. [Online]. Available: <https://fastapi.tiangolo.com/>
- [9] React. [Online]. Available: <https://react.dev/>
- [10] Redux. [Online]. Available: <https://redux.js.org/>
- [11] Vis. [Online]. Available: <https://visjs.github.io/vis-network/docs/network/>
- [12] Docker. [Online]. Available: <https://docs.docker.com/>