DATA STRUCTURE & ALGORITHMS
CSE271S

XML EDITOR APP

SUBMITTED TO: DR. ISLAM ELMADAH
ENG. FADY FARAGALLAH

DATE OF SUBMISSION: 24TH DEC. 2025

FACULTY OF ENGINEERING
AIN SHAMS UNIVERSITY

Contents

Name	**ID**
Amr Ahmed Wahidi	**2200429**
Ali Ahmed Mohamed Abdelmaboud	**2200927**
Hamza Basem Mohamed Ahmed	**2200496**
Omar Mohamed Hassan	**2200294**
Ammar Mohamed Abdelghany	**2200707**
Mohamed Elsayed Moustafa	**2200095**
Ahmed Ramadan	**2200372**
Anas Ayman Mohammed	**2200325**
Ahmed Ezzat Ibrahim Hammad	**2201472**

1. Background

1.1 Introduction

In the modern landscape of data interchange, XML (Extensible Markup Language) remains a cornerstone format for storing and sharing structured information across diverse platforms. While robust, raw XML files can be verbose and difficult to manage manually. This project, the XML Editor, was developed to address these challenges by creating a comprehensive desktop application capable of parsing, visualizing, and analyzing XML data.

Beyond simple text processing, this system interprets the XML specifically as a dataset for a social network. The input files represent users, their posts, and their followers. This dual-purpose application serves not only as a file manipulation tool (checking consistency, formatting, compressing) but also as a network analysis engine, employing Graph theory to uncover insights like user influence and connection suggestions.

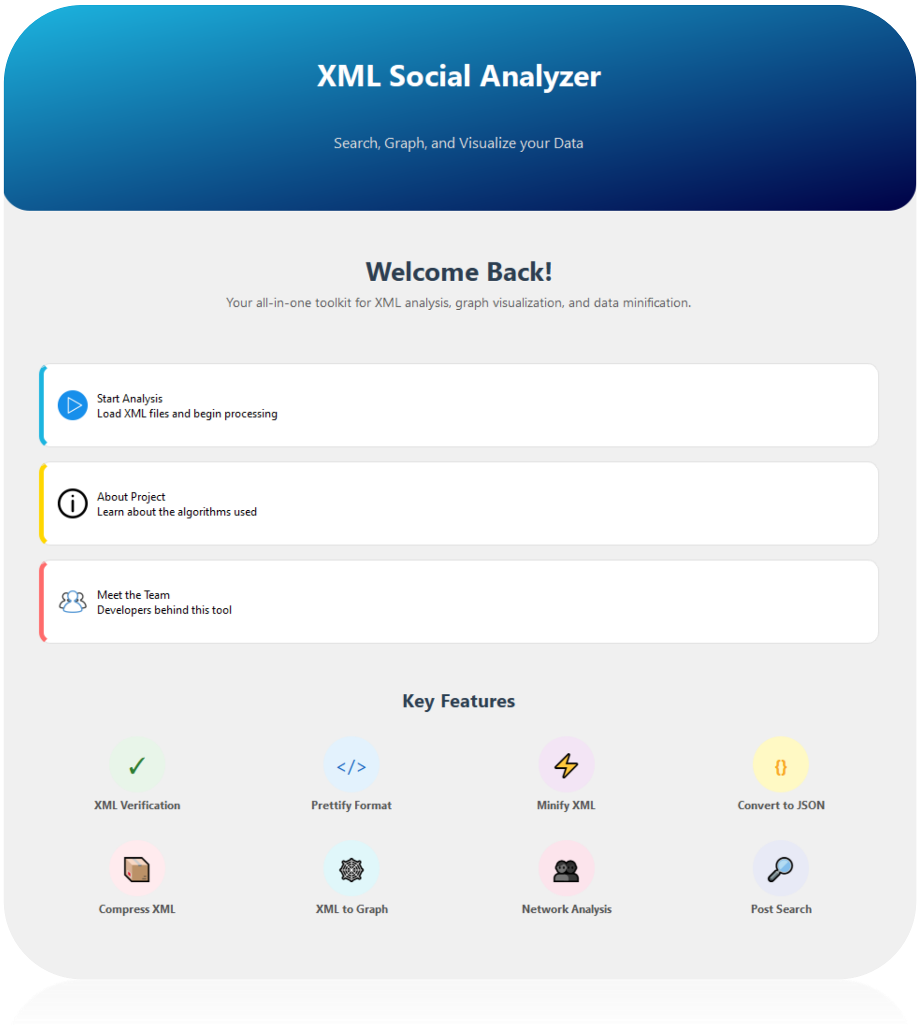
1.2 System Overview

The application operates in two distinct modes to cater to different user needs:

GUI Mode: A user-friendly graphical interface allowing file browsing, visual output of operations, and interactive graph drawing.

Command Line Mode (CLI): A streamlined interface for power users to execute specific operations (e.g., xml_editor verify -i input.xml) efficiently.

Installation & Deployment: To ensure seamless deployment, we authored a custom **Inno Setup Script (.iss)** to define the installer's logic and configured **CMake** to automatically bundle all necessary **.dll** dependencies directly into the build output. This process created a self-contained directory to eliminate runtime errors, which was then compiled into a single, standalone installation file for easy distribution.

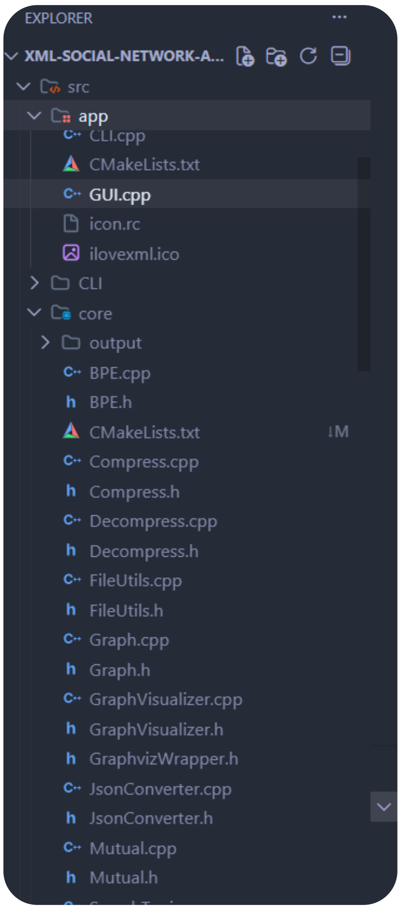

XML Social Analyzer
Search, Graph, and Visualize your Data
Welcome Back!
Your all-in-one toolkit for XML analysis, graph visualization, and data minification.
Start Analysis
Load XML files and begin processing
About Project
Learn about the algorithms used
Meet the Team
Developers behind this tool
Key Features
XML Verification
Prettify Format
Minify XML
Convert to JSON
Compress XML
XML to Graph
Network Analysis
Post Search

Technical Highlights:

- Utilizes **stacks** for XML verification, **graphs** for network analysis and visualization, **tree** for converting, and **hash maps** for efficient compression and decompression.
- Dual interface (GUI + CLI) ensures accessibility for both casual and advanced users.

Git Architecture & Project Structure:

- The project is version-controlled using **Git**, with the repository organized into multiple directories for clarity and maintainability:
  - src/ – Contains all the source code of the application.
  - installation/ – Includes installation scripts and dependencies for setting up the application.
  - .gitignore – Specifies files and folders to be excluded from version control.
  - CMakeLists.txt – Build configuration file for compiling the project.
  - README.md – Serves as a **setup guide**, providing instructions on how to install, build, and run the application.



```
EXPLORER
XML-SOCIAL-NETWORK-A...
  src
    app
      CLI.cpp
      CMakeLists.txt
      GUI.cpp
      icon.rc
      ilovexml.ico
    CLI
    core
      output
      BPE.cpp
      BPE.h
      CMakeLists.txt        M
      Compress.cpp
      Compress.h
      Decompress.cpp
      Decompress.h
      FileUtils.cpp
      FileUtils.h
      Graph.cpp
      Graph.h
      GraphVisualizer.cpp
      GraphVisualizer.h
      GraphvizWrapper.h
      JsonConverter.cpp
      JsonConverter.h
      Mutual.cpp
      Mutual.h
```

- Each commit is documented with meaningful messages to track progress, bug fixes, and feature updates.

By combining **robust functionality, advanced analytics, and a well-structured Git architecture**, the project ensures both usability and maintainability for development and deployment.

1.3 Demo Video

[Click here for Demo Video]

2. Implementation Details of App Algorithms

This section details the architectural choices and specific algorithms employed to handle the project requirements. A core constraint of this project was the requirement to implement complex data structures (specifically Trees and Graphs) from scratch, while utilizing standard libraries only for basic linear structures like stacks and vectors.

2.1 Custom Data Structure

To comply with project constraints, several fundamental data structures were implemented from scratch:

- **General Tree (Tree.h)**: A custom template-based tree structure where each Node contains a data element and a std::vector of pointers to its children. This is the intermediate representation used for XML prettifying and JSON conversion.
- **Adjacency List Graph (Graph.h)**: The social network is represented using a std::map<int, std::vector<int>>, where keys are unique User IDs and values are adjacency lists of followers.
- **Simple Map (SimpleMap.h)**: A custom mapping structure used within the compression logic to track frequency pairs.

2.2 Verification and Correction

The structural integrity of the XML files is managed by the XMLValidator.cpp module, which uses a combination of stack-based scanning and error-tracking structures.

Verification Algorithm

The verification algorithm checks whether an XML document is well-formed by validating the correct structure of tags.
It scans the XML content character by character and uses a stack-based approach to ensure proper nesting and matching of tags.

- **Stack-Based Parsing**: The verify function iterates through the XML string using a std::stack<std::string> to track nesting levels.
- **Tag Identification**: When the parser encounters an opening tag (e.g., <user>), the tag name is extracted and pushed onto the stack.
- **Matching Logic**: When a closing tag (e.g., </user>) is found, the algorithm checks if the stack is empty or if the top of the stack matches the current closing tag.
- **Validation**: If the tags match, the top element is popped; if they do not match, or if the stack is not empty at the end of the file, the XML is marked as invalid.
- **Self-Closing Tags**: The algorithm includes logic to detect self-closing tags (e.g., <img />), which are identified without being pushed onto the stack, preserving the correct nesting depth.

Main Steps:

1. **Scan the input sequentially** from left to right.
2. **Detect opening tags**, closing tags, and self-closing tags.
3. **Validate tag names**:
    - The first character must be a letter.
    - Remaining characters may include letters, digits, or special characters (- _ . :).
4. **Handle tags using a stack**:
    - Push opening tags onto the stack.
    - Pop the stack when a matching closing tag is found.
5. **Reject invalid cases**, such as:
    - Mismatched closing tags.
    - Missing > character.
    - Attributes in closing tags.
    - Invalid characters in tag names.

6. **Ignore text content** between tags safely.
7. At the end, the XML is valid **only if the stack is empty**.

Result:

- If all tags are properly matched and nested → **Valid XML**
- Otherwise → **Invalid XML**

Correction Algorithm

- **Error Summary**: The countErrorSummary function performs a pre-pass to identify exactly where inconsistencies occur. It uses an ErrorInfo struct to store the line number, nesting level, and tag name associated with each error.
- **Automated Fixing**: The fixXml function iterates through the original content and uses the ErrorInfo data to reconstruct the file.
- **Tag Insertion**: If a closing tag is missing, the algorithm calculates the correct indentation (level * 4 spaces) and appends the necessary closing tag (e.g., </user>) at that position.
- **Mismatched Handling**: For mismatched tags, it identifies the expected tag from the internal stack and replaces or appends the correct structural element to restore validity.

Analysis

- The verification algorithm is strict and precise.
  It does not attempt to guess or repair errors; it only checks correctness.
- The correction algorithm is heuristic-based.
  It tries to infer the programmer's intention and fix structural mistakes, mainly related to missing or mismatched closing tags.
- Both algorithms rely heavily on stack-based parsing, which is ideal for hierarchical structures like XML.

Limitations

Correction Algorithm Limitations:

- Cannot reliably fix deeply ambiguous structures.
- May fail when:
    - Opening tags are completely missing.
    - Tag hierarchy is severely broken.
- Assumes indentation and nesting levels reflect logical structure.
- Focuses mainly on closing-tag-related errors.

Verification Algorithm Strength:

- The verification algorithm has no structural limitations.
- It correctly detects all malformed XML cases, including:
    - Invalid characters
    - Incorrect nesting
    - Missing or extra tags
- It is robust, strict, and reliable, but intentionally does not perform auto-**correction**.

Time Complexity is O(n)

Space Complexity is O(n)

2.3 Formatting and Minification

Prettifying Algorithm (XML Formatting)

The Prettifying algorithm transforms a raw, single-line, or poorly formatted XML string into a structured, human-readable format with consistent indentation.

- **Tree Construction**: The algorithm first parses the XML into a custom **General Tree** structure. Each XML tag becomes a Node, and nested tags become its children.
- **Recursive Traversal**: The core logic resides in print_prettified_helper, which performs a depth-first traversal of the tree.
- **Indentation Management**: A string variable (usually three or four spaces) is passed through each recursive call. Every time the algorithm moves to a child node, it increases the indentation level.
- **Leaf Wrapper Optimization**: Your implementation includes a specific check for "leaf wrappers"—nodes that contain only one child which is a text value. Instead of splitting these onto three lines, the algorithm keeps the opening tag, the text, and the closing tag on a single indented line (e.g., <name>Anas</name>) for better readability.
- **Closing Tags**: As the recursion unwinds, the algorithm appends the closing tag at the exact same indentation level as the opening tag to ensure perfect vertical alignment.

XML Minifier

The XML minifier algorithm reduces the size of an XML document by removing unnecessary whitespace while preserving the structural meaning of the tags and content.

- It processes the input string character by character in a single pass.
- A flag called insideTag tracks whether the current position is within an XML tag or outside in text content.

- When inside a tag:
    - All characters are preserved except formatting symbols such as newlines, carriage returns, and tabs, which are discarded.
- When outside a tag:
    - The algorithm collapses multiple consecutive spaces into a single space.
    - It removes trailing spaces before a tag begins.
- Another flag, prevSpace, remembers whether the last character added was a space, ensuring that duplicate spaces are not introduced.
- The algorithm appends characters to the result string according to these rules, producing a compact version of the XML that is semantically identical to the original but free of redundant formatting.

Complexity Analysis

- **Time Complexity**: $O(N)$, where $N$is the number of characters in the XML string, since each character is processed in constant time.
- **Space Complexity**: $O(N)$, because the output string is built to contain nearly the same number of characters as the input (minus removed whitespace), and temporary flags require only constant additional memory.

2.4 Conversion and Search

Data Conversion (XML to JSON)

The **JsonConverter** module employs a recursive Depth-First Search (DFS) algorithm to transform the internal tree representation of the XML into a valid JSON string. This conversion handles the structural differences between the two formats, specifically addressing the challenge of identifying JSON arrays, which are not explicitly defined in XML.

Conversion Logic

- **Recursive Traversal:** The convertToJson function traverses the Node<string> tree. For each node, it determines whether to render a simple key-value pair, a nested object, or a JSON array.
- **Leaf Node Handling:** The isLeafWrapper helper function identifies "simple" XML elements (tags containing only text, e.g., <id>1</id>). These are converted directly into JSON string values (e.g., "id": "1") to prevent unnecessary nesting.
- **Array vs. Object Detection:** A core feature of the algorithm is its ability to automatically group repeated tags into JSON arrays.
    - Instead of using a standard hash map, the function utilizes two synchronized vectors: uniqueTags (to track the order and frequency of tag names) and grouped (to store pointers to the corresponding child nodes).
    - As the algorithm iterates through a node's children, it checks if a tag name has been encountered before.
    - **Single Occurrence:** If a tag appears only once, it is rendered as a standard JSON object property.
    - **Multiple Occurrences:** If a tag appears multiple times (e.g., multiple <post> tags inside <posts>), the algorithm detects the repetition via the uniqueTags counter and renders the children as a JSON array [...].

Complexity Analysis

Time Complexity: O(N * K) (Average Case: O(N))

- The algorithm visits every node in the XML tree (N) exactly once during the traversal.
- *Local Overhead:* For each node, the algorithm iterates through its children to group them. In the worst-case scenario (a node with many unique children), searching the uniqueTags vector takes time proportional to the number of children (K). However, in typical XML structures where children are either few in number or repeated

identical tags (arrays), this operation is highly efficient, making the overall complexity effectively linear, **O(N)**.

Space Complexity: O(N)

- **Auxiliary Space:** The vectors uniqueTags and grouped consume memory proportional to the branching factor of the tree, which sums to O(N) across the entire execution.
- **Output Storage:** The resulting JSON string requires O(N) space to store the converted characters.
- **Stack Space:** The recursion depth depends on the height of the XML tree (H), contributing O(H) to the stack memory usage.

Word/Topic Search

The word and topic search algorithms both operate by scanning the XML document as a string and applying a structured traversal pattern.

- The process begins by locating <user> blocks, which represent individual accounts in the dataset.
- For each user block, the algorithm extracts the <id> tag to identify the author.
- It then iterates through all <post> elements contained within that user. Each post is isolated by slicing between <post> and </post> delimiters.
- Helper functions normalize the extracted content:
    - toLower ensures case-insensitive comparisons.
    - trim removes leading and trailing whitespace.
    - extractTagContent retrieves the inner text of specific tags.

In the word search variant, the algorithm inspects <body> or <text> tags. In the topic search variant, it inspects <topic> tags.

In both cases, the search condition is a substring check:

- The query string is converted to lowercase and compared against the candidate text.

- If the query appears anywhere inside, the post is marked as a match.
- Each match is stored in a PostMatch object that links the user's identifier with the relevant post content.

Complexity Analysis

- **Time Complexity**: $O(N)$, where $N$is the length of the XML string. Each character is scanned once, and substring checks scale linearly with the length of post text.
- **Space Complexity**: $O(M + K)$, where $M$is the size of temporary substrings extracted during parsing and $K$is the number of matches stored in the result vector.

2.5 Social Network Analysis

Most Influencer Algorithm (getMostInfluencerId())

Algorithm Explanation: This algorithm identifies the user(s) who follow the most people in the network. It performs a two-pass approach over the graph data structure. In the first pass, it iterates through all users to find the maximum outdegree (number of people a user follows). The second pass collects all user IDs whose outdegree equals this maximum value, allowing for multiple users to be identified if there's a tie.

Time Complexity: O(n), where n is the number of users. The algorithm makes two sequential passes through the graph map, each taking linear time.

Space Complexity: O(k), where k is the number of users with the maximum outdegree. In the worst case where all users have the same outdegree, k = n, making it O(n).

Most Active Person Algorithm (getMostActivePersonId())

Algorithm Explanation: This algorithm determines the user(s) with the highest indegree (most followers). It first initializes a map to store following counts for all users, setting initial values to zero. Then, it iterates through the entire graph, examining each user's follower list and incrementing the count for each follower ID found. After counting, it performs two passes: one to find the maximum follower count, and another to collect all user IDs matching this maximum. This identifies users who are followed by the most people in the network.

Time Complexity: O(n + e), where n is the number of users and e is the total number of edges (follower relationships). The initialization takes O(n), counting followers requires traversing all edges O(e), and finding the maximum requires two additional O(n) passes.

Space Complexity: O(n) for the followingCounts map that stores indegree information for every user, plus O(k) for the result vector, where k ≤ n is the number of users with maximum indegree.

Both algorithms prioritize correctness and handle tie scenarios by returning all qualifying users rather than arbitrarily selecting one.

Suggest Algorithm (Follower Recommendations)

The Suggest algorithm identifies potential new connections for a user based on the "Followers of Followers" social logic.

- **Graph Representation**: The algorithm utilizes the **Adjacency List** stored in the Graph class, where each user ID is mapped to a list of followers.
- **Step 1: Identify Direct Connections**: For a target user (User A), the algorithm first retrieves their list of followers (Users B, C, D).
- **Step 2: Traverse to Second-Degree Connections**: The algorithm iterates through each of those followers and retrieves *their* followers (the "followers of followers").
- **Step 3: Aggregation and Filtering**:
    - **Collection**: All identified second-degree connections are added to a potential suggestion list.
    - **Uniqueness**: The list is sorted and processed to remove duplicate IDs.

  - **Self-Exclusion**: The algorithm explicitly removes User A’s own ID from the suggestions so they aren't suggested to follow themselves.
  - **Relationship Filtering**: Finally, the removeValues helper function compares the suggestions against User A's current "following" list. It removes anyone User A already follows, ensuring the final list only contains brand-new potential connections.

Mutual Followers

The mutual followers algorithm computes the set of accounts that follow all users in a given list of IDs by combining graph adjacency queries with set intersection operations.

- It begins by retrieving the follower list of the first user from the graph structure using getNeighbors.
- This list is sorted to prepare for intersection.
- For each additional user ID:
  - The algorithm retrieves that user’s follower list.
  - Sorts it.
  - Computes the intersection between the current result set and the new list using the standard library’s set_intersection function.
- This intersection step progressively reduces the candidate set until only those followers that appear in every list remain.
- The final result is a vector containing the IDs of mutual followers.
- The helper function printVector formats the output, either displaying the list of IDs or reporting that none were found.

Complexity Analysis

- **Time Complexity**: $O(u\log\ u + u)$, where $u$is the size of the follower list for each user.
  - Each list must be sorted before intersection.

   - The intersection itself runs in linear time relative to the list sizes.
   - Overall complexity simplifies to $O(u\log\ u)$.
- **Space Complexity**: $O(R+V)$, where $R$is the size of the intersection result and Vaccounts for temporary vectors created during sorting and intersection.

2.6 Data Compression (Byte Pair Encoding)

For the data compression component of this project, we implemented **Byte Pair Encoding (BPE)** to optimize the storage of XML and JSON files. BPE is a lossless compression algorithm that reduces file size by iteratively identifying and replacing the most frequent patterns in the data with shorter, unique symbols. We selected BPE over other compression options due to its algorithmic clarity, and its effectiveness.

Algorithm

The BPE algorithm treats the input text as a sequence of tokens. Initially, every individual byte is considered a token. The compression process follows an iterative cycle:

1. **Frequency Analysis:** The system scans the input text to build a frequency table of all adjacent byte pairs. To optimize performance, we utilized a custom hash map (SimpleMap). This allows for constant-time access (O(1)) during frequency counting, which is significantly faster than linear searches or array-based lookups.
2. **Selection and Merge:** The algorithm identifies the single most frequently occurring adjacent pair. This pair is "merged" by replacing all its occurrences with a new, unused byte token (values starting from ASCII 128).
3. **Dictionary Management:** The mapping between the new token and the original pair is stored in a dictionary vector. This is essential for the eventual reconstruction of the data.

4. **Iteration:** Steps 1 through 3 are repeated until a target vocabulary size is reached or no pair appears frequently enough to warrant compression.

The final compressed output consists of a file type indicator, the reconstruction dictionary, and the encoded text stream.

Decompression

Decompression is the exact inverse of the compression process. Because BPE is lossless, it guarantees full recovery of the original XML structure. The system reads the dictionary and expands tokens back into their original byte pairs. Crucially, this expansion is performed in **reverse order** (Last-In-First-Out). This ensures that complex tokens—which may be composed of other tokens created earlier in the process—are expanded correctly without dependency conflicts.

Complexity Analysis

- **Time Complexity:** Compression operates at **O(I * n)**, where n is the text length and I is the number of iterations (bounded by the token limit). Decompression operates at **O(d * n)**, where d is the number of dictionary entries.
- **Space Complexity:** Both phases maintain linear space complexity **O(n)**, making the solution memory-efficient and scalable for larger input files.

For detailed C++ implementation details with test cases:

https://drive.google.com/drive/folders/1p2knvwryk40c1ft5_iSsW7o_1b2Wiu4d?usp=sharing

2.7 Graph Visualization

The project visualizes the social network by transforming raw XML data into a graphical format using the **Graphviz** engine.

- **Adjacency List Construction**: The Graph class first parses the XML to build an internal representation where each user ID is a node and their followers are stored in a std::map<int, std::vector<int>>.
- **DOT Language Generation**: The GraphVisualizer.cpp module iterates through this adjacency list to generate a string in the **DOT language**. This format describes the graph structure (e.g., User1 -> User2;).
- **Node Customization**: During generation, the algorithm maps unique user IDs to their respective names (stored in a names map) to ensure nodes are labeled correctly in the final image.
- **Rendering Bridge**: The system uses a GraphvizWrapper (or bridge) to send the generated DOT string to the Graphviz library.
- **Image Output**: The visualization engine processes the directed relations and calculates the optimal layout (using algorithms like *dot* or *neato*) to produce a .jpg or .png file representing the network.

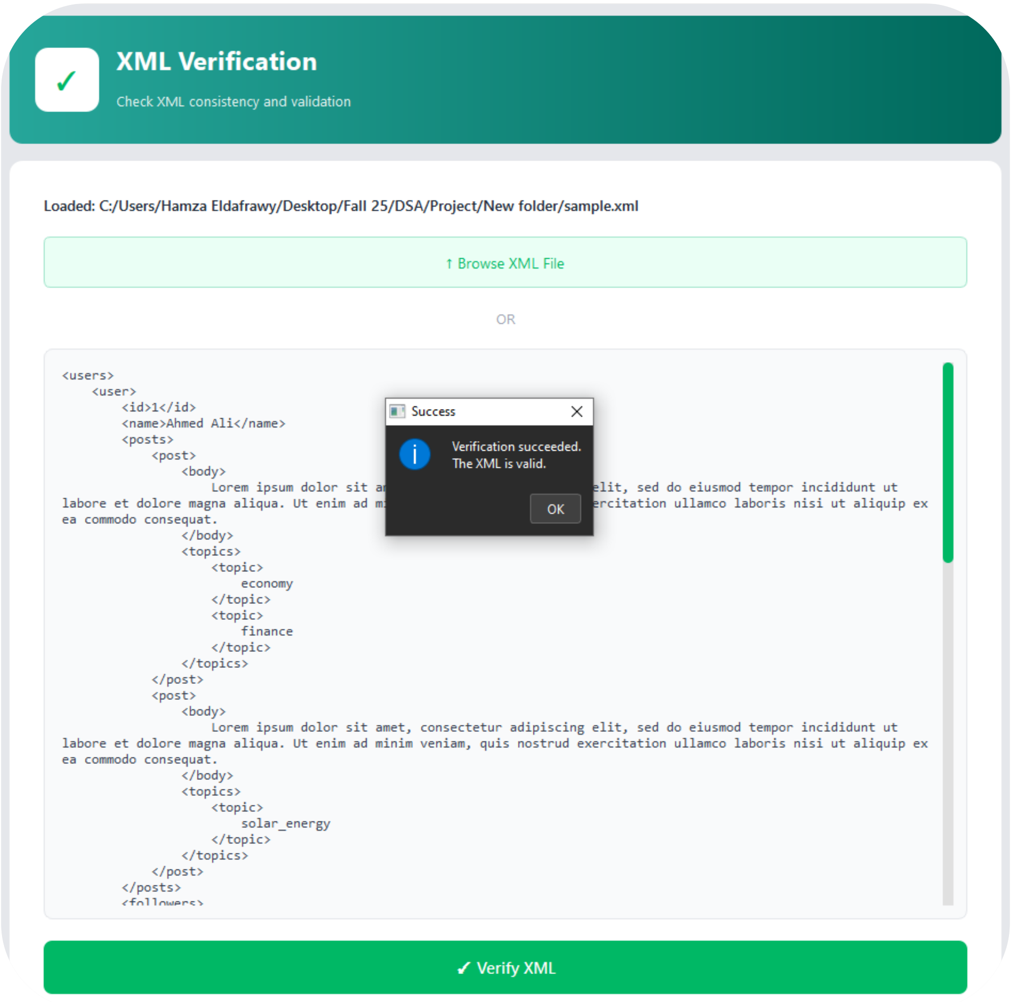
Space Complexity is O(u+e) for Build Graph and Visualization

3. Test walkthrough

3.1 XML Verification and Correction

The XMLValidator.cpp module ensures the structural integrity of input files through two primary methods:

- **Verification**: A stack-based algorithm scans XML tokens. Opening tags are pushed onto a std::stack, and when a closing tag appears, it is compared with the stack's top element; mismatches or empty stacks indicate an invalid structure.
- **Error Correction (fixXml)**: The system identifies specific lines with missing or mismatched tags and automatically inserts the correct closing tags based on the current nesting level and the last opened tag.



Loaded: C:/Users/Hamza Eldafrawy/Desktop/Fall 25/DSA/Project/New folder/sample.xml

```
<users>
    <user>
        <id>1</id>
        <name>Ahmed Ali</name>
        <posts>
            <post>
                <body>
                    Lorem ipsum dolor sit a                     elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad m                     ercitation ullamco laboris nisi ut aliquip ex
ea commodo consequat.
                </body>
                <topics>
                    <topic>
                        economy
                    </topic>
                    <topic>
                        finance
                    </topic>
                </topics>
            </post>
            <post>
                <body>
                    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo consequat.
                </body>
                <topics>
                    <topic>
                        solar_energy
                    </topic>
                </topics>
            </post>
        </posts>
        <followers>
```

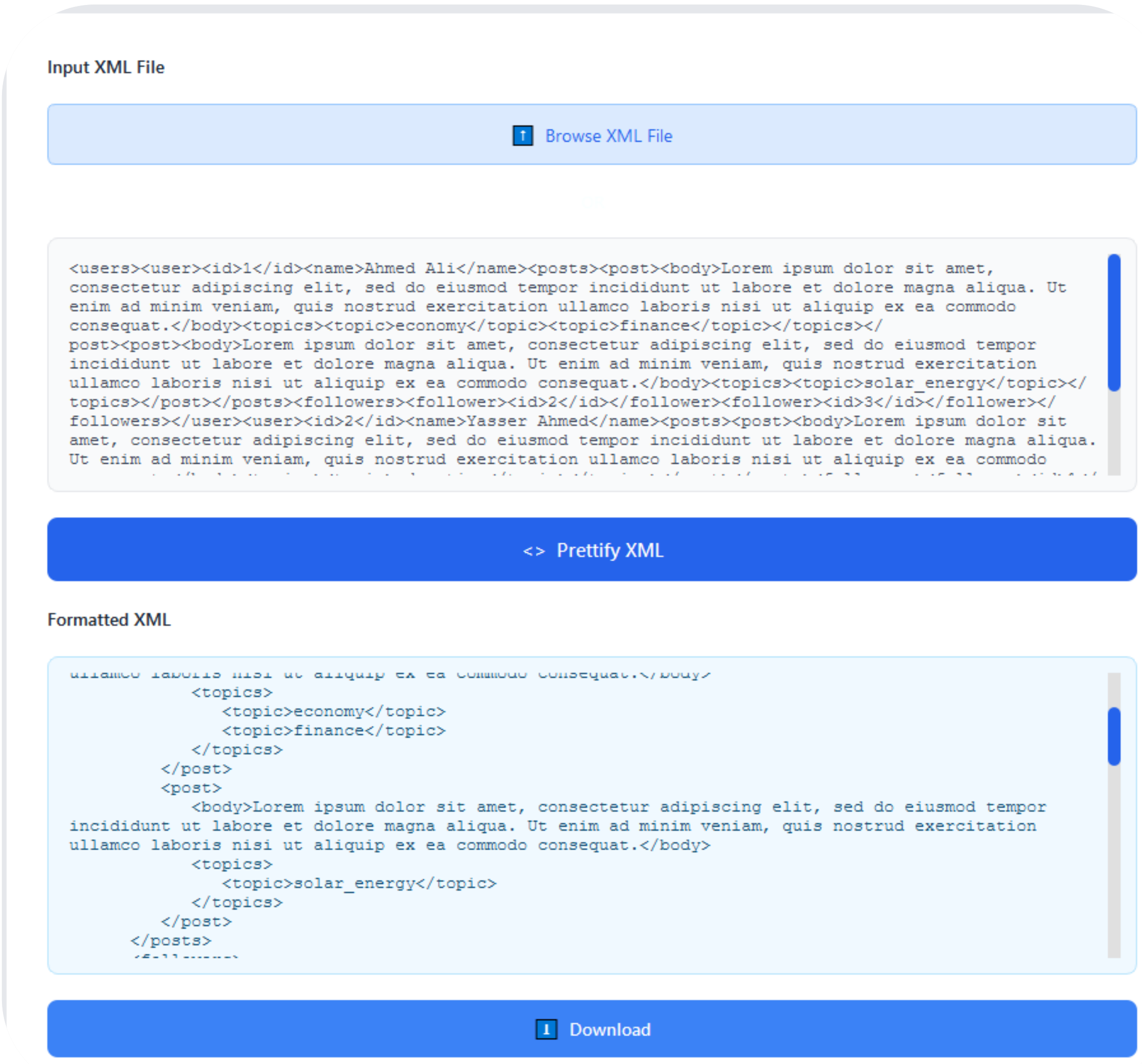
Success

Verification succeeded.
The XML is valid.

OK

3.2 Formatting and Minification

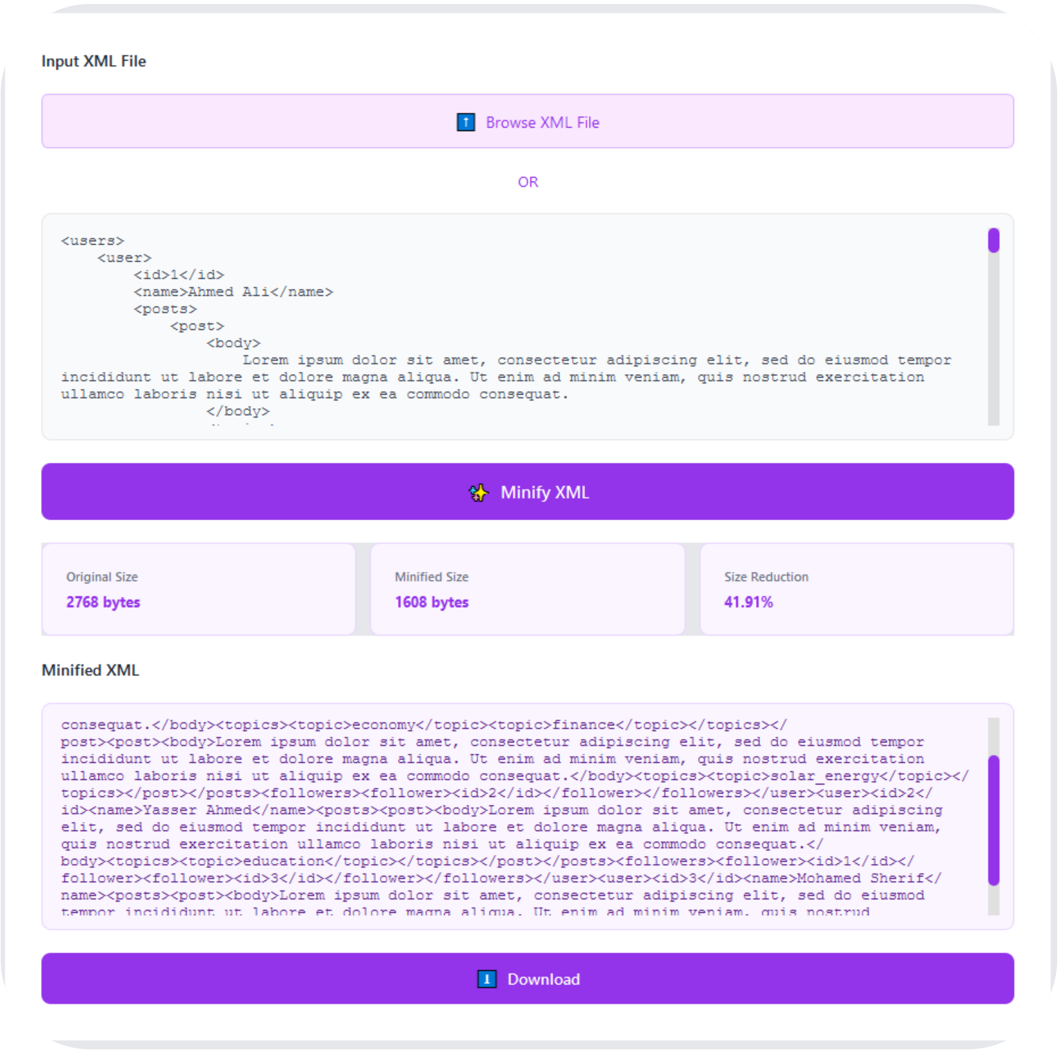
- **Formatting (Prettifying)**: Utilizing the custom Tree structure, the system performs a recursive traversal. It identifies "leaf wrappers"—tags containing only text—and formats them on single lines, while applying consistent indentation to deeper nested levels.



```
<users><user><id>1</id><name>Ahmed Ali</name><posts><post><body>Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat.</body><topics><topic>economy</topic><topic>finance</topic></topics></
post><post><body>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.</body><topics><topic>solar_energy</topic></
topics></post></posts><followers><follower><id>2</id></follower><follower><id>3</id></follower></
followers></user><user><id>2</id><name>Yasser Ahmed</name><posts><post><body>Lorem ipsum dolor sit
amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
```

```
ullamco laboris nisi ut aliquip ex ea commodo consequat.</body>
            <topics>
               <topic>economy</topic>
               <topic>finance</topic>
            </topics>
         </post>
         <post>
            <body>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.</body>
            <topics>
               <topic>solar_energy</topic>
            </topics>
         </post>
      </posts>
```

- **Minifying**: The XMLMinifier.cpp logic reduces file size by stripping unnecessary newlines, tabs, and extra spaces. It tracks whether the pointer is "inside" or "outside" a tag to preserve critical content while collapsing redundant whitespace.



```
<users>
    <user>
        <id>1</id>
        <name>Ahmed Ali</name>
        <posts>
            <post>
                <body>
                    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.
                </body>
```

Original Size
2768 bytes

Minified Size
1608 bytes

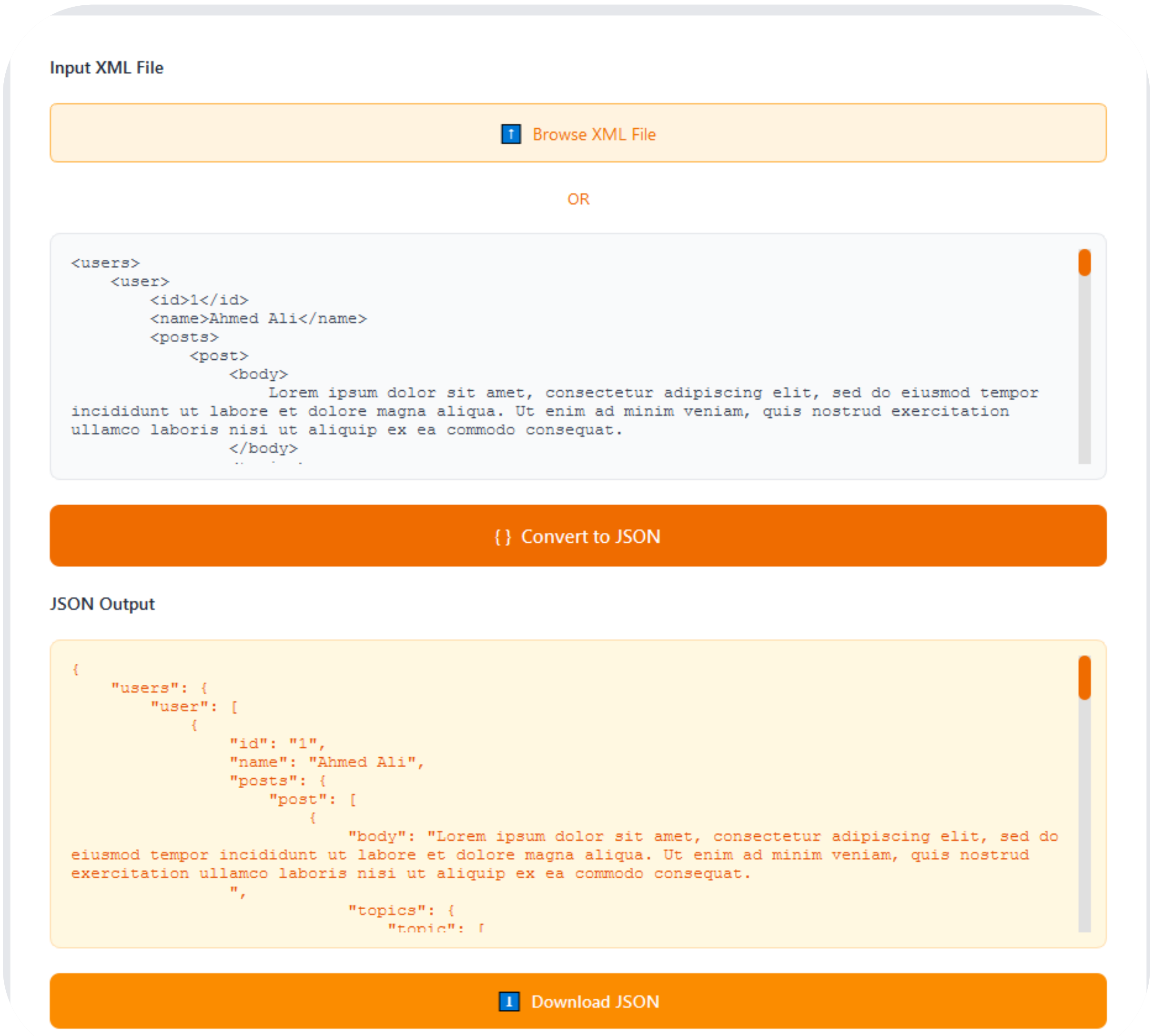
Size Reduction
41.91%

Minified XML

```
consequat.</body><topics><topic>economy</topic><topic>finance</topic></topics></
post><post><body>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.</body><topics><topic>solar_energy</topic></
topics></post></posts><followers><follower><id>2</id></follower></followers></user><user><id>2</
id><name>Yasser Ahmed</name><posts><post><body>Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.</
body><topics><topic>education</topic></topics></post></posts><followers><follower><id>1</id></
follower><follower><id>3</id></follower></followers></user><user><id>3</id><name>Mohamed Sherif</
name><posts><post><body>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
```

3.3 Data Conversion and Search

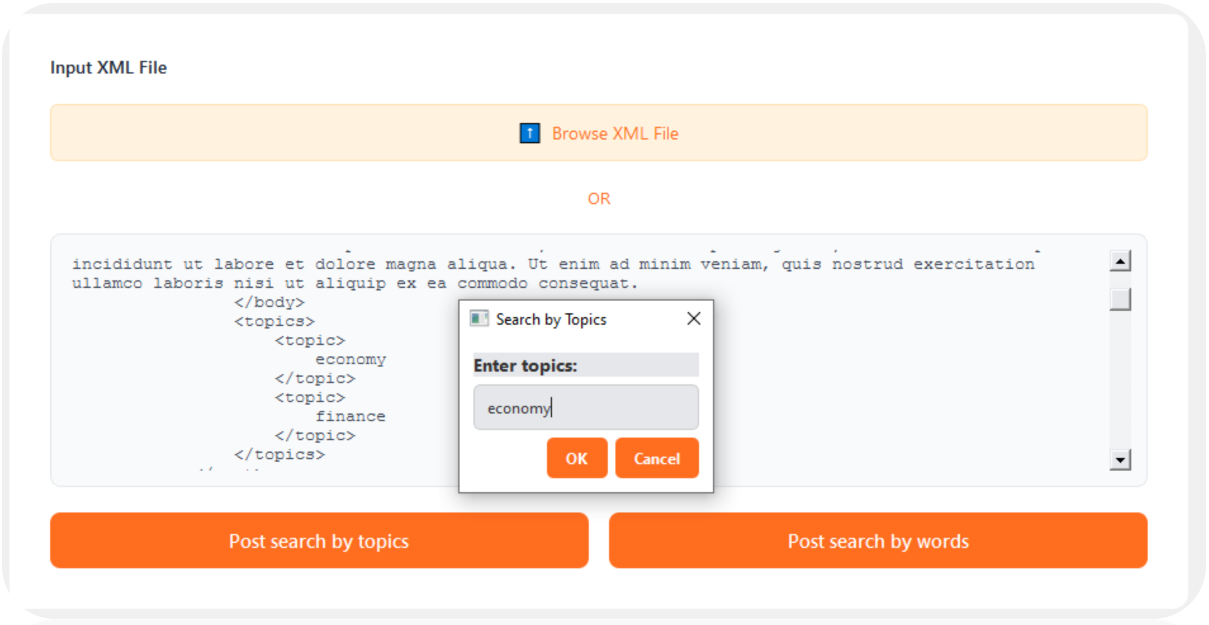
- **XML to JSON Conversion**: The JsonConverter recursively traverses the XML tree. It features advanced array detection logic that groups repeated tags (e.g., multiple <post> entries) into single JSON arrays [] rather than duplicate keys.



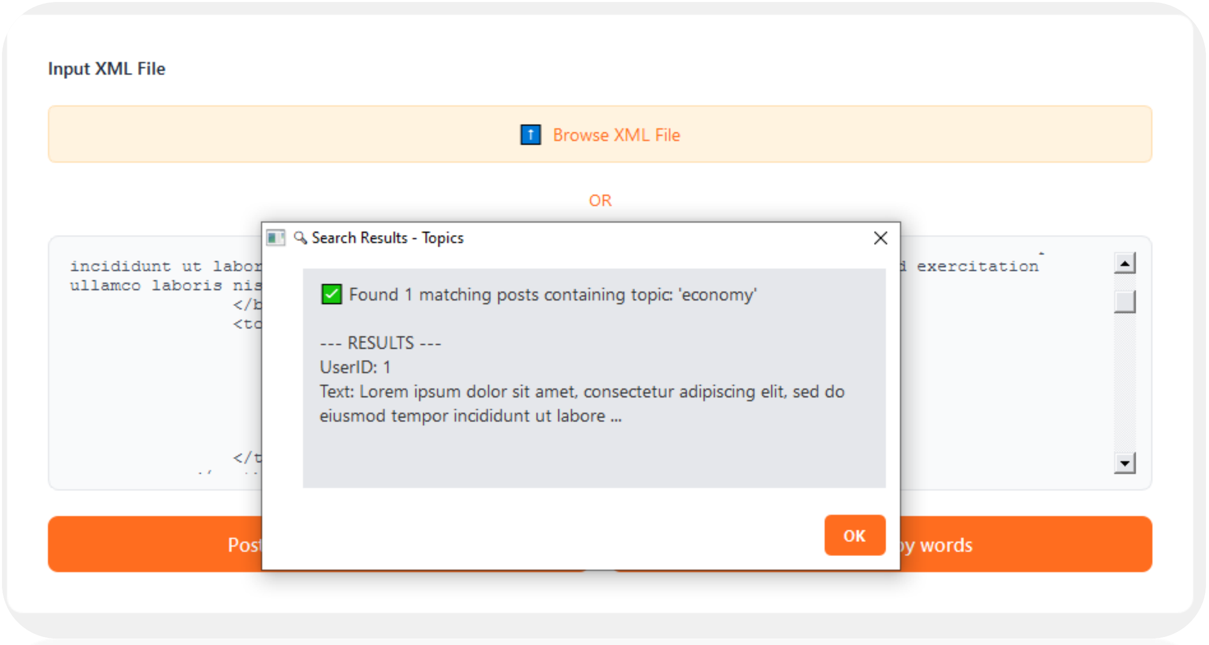
```
<users>
    <user>
        <id>1</id>
        <name>Ahmed Ali</name>
        <posts>
            <post>
                <body>
                    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.
                </body>
```

```
{
    "users": {
        "user": [
            {
                "id": "1",
                "name": "Ahmed Ali",
                "posts": {
                    "post": [
                        {
                            "body": "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
                ",
                            "topics": {
```

- **Post Search**: Dedicated modules SearchWord.cpp and SearchTopic.cpp allow the system to iterate through user post lists to find specific keywords or topics mentioned in the network.



```
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.
                </body>
                <topics>
                    <topic>
                        economy
                    </topic>
                    <topic>
                        finance
                    </topic>
                </topics>
```



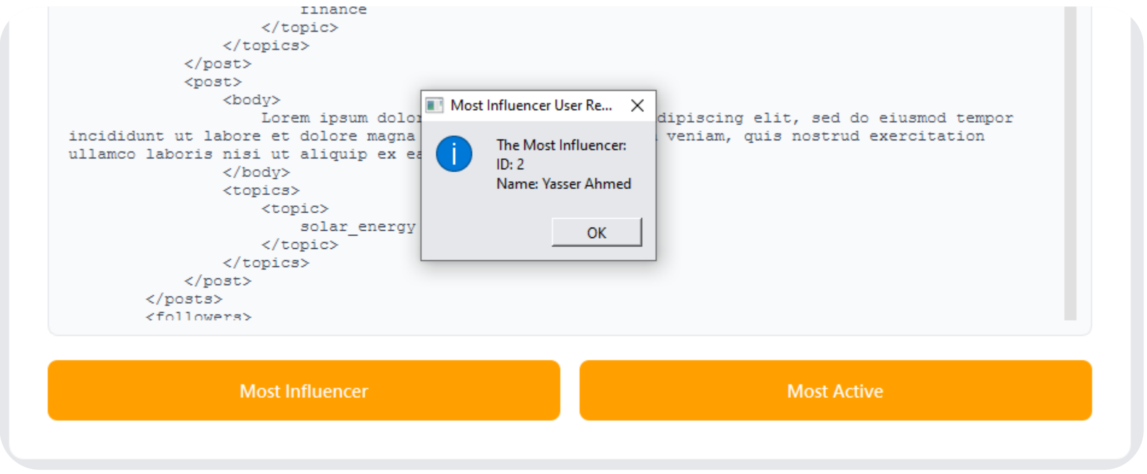
```
✅ Found 1 matching posts containing topic: 'economy'

--- RESULTS ---
UserID: 1
Text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore ...
```

3.4 Social Network Analysis

Using the Graph class, the system extracts critical network data:

- **Influence & Activity**: "Most Influencer" is identified by calculating the highest **in-degree** (most followers), while the "Most Active" is determined by **out-degree** or total post count.

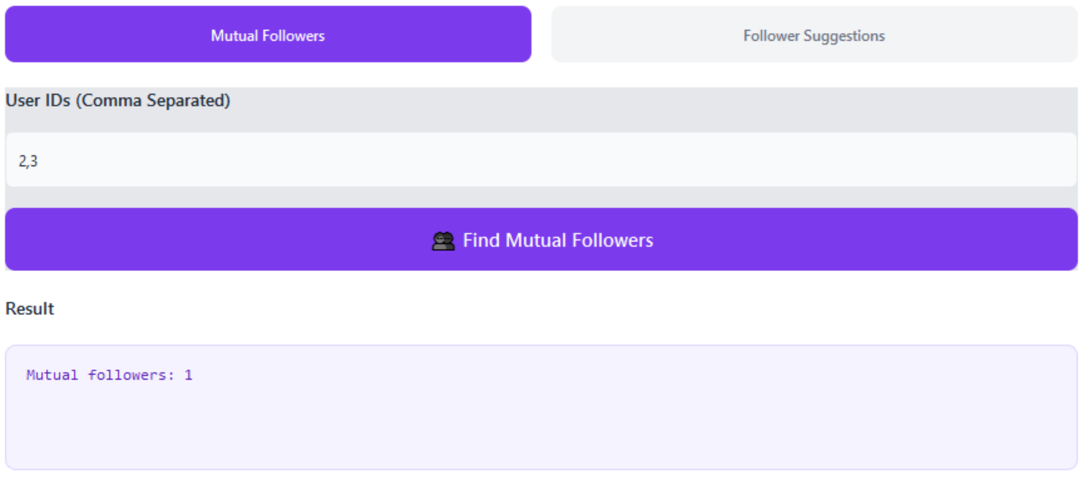


```
                    finance
                </topic>
            </topics>
        </post>
        <post>
            <body>
                Lorem ipsum dolo                dipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna              veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea
            </body>
            <topics>
                <topic>
                    solar_energy
                </topic>
            </topics>
        </post>
    </posts>
    <followers>
```

Most Influencer User Re...

The Most Influencer:
ID: 2
Name: Yasser Ahmed

- **Mutual Followers**: The Mutual.cpp module retrieves follower lists for multiple users and uses std::set_intersection on sorted vectors to identify common connections.
- **Follower Suggestions**: The Suggest.cpp module implements a "followers of followers" algorithm, identifying users connected to a user's current followers while filtering out existing connections and the user themselves.



User IDs (Comma Separated)

2,3

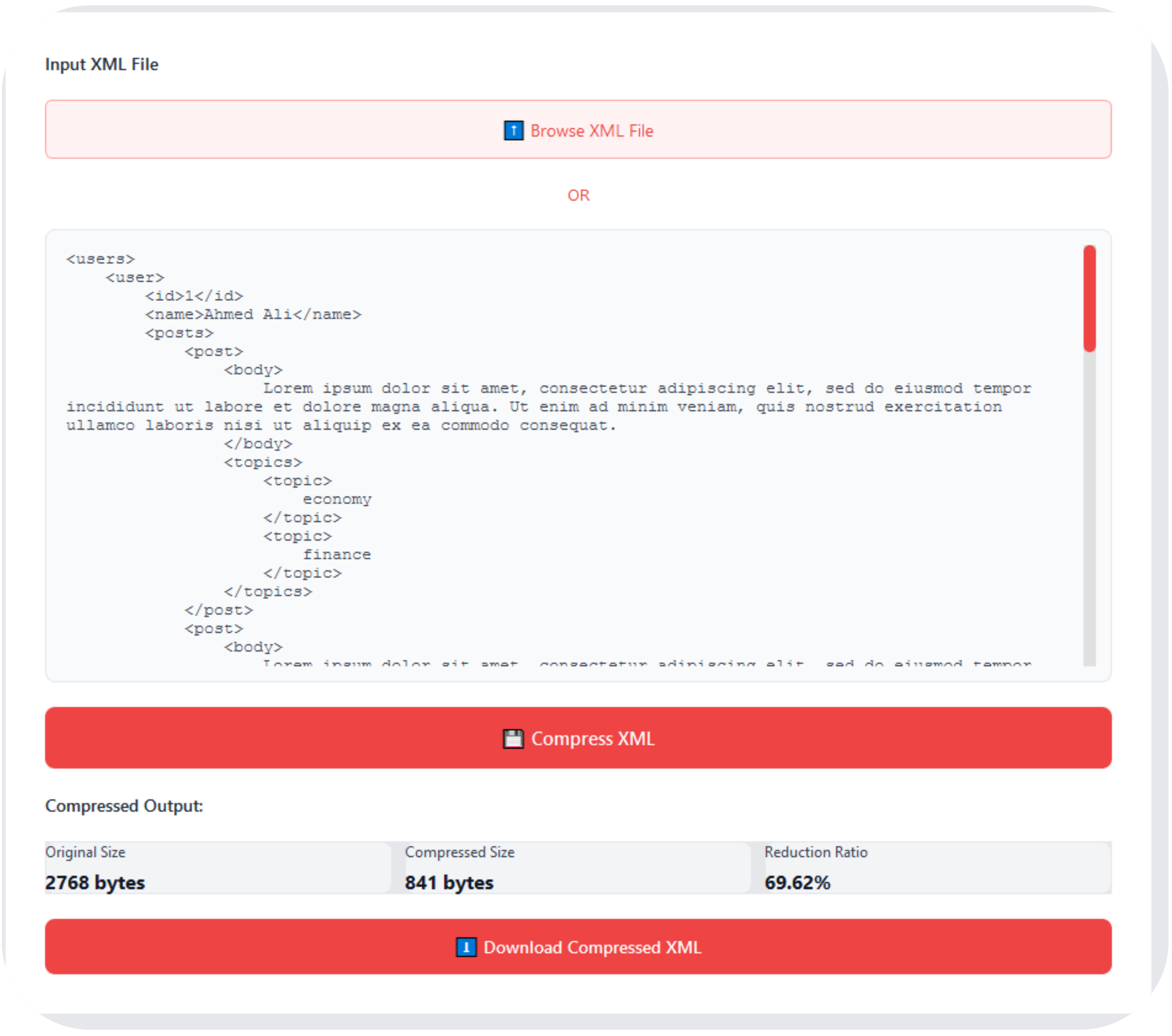
Result

```
Mutual followers: 1
```

3.5 Data Compression (Byte Pair Encoding)

The project utilizes **Byte Pair Encoding (BPE)** for compression.

- **Compression**: The algorithm builds a frequency table of character pairs and replaces the most frequent pair with an unused byte (starting from ASCII 128).



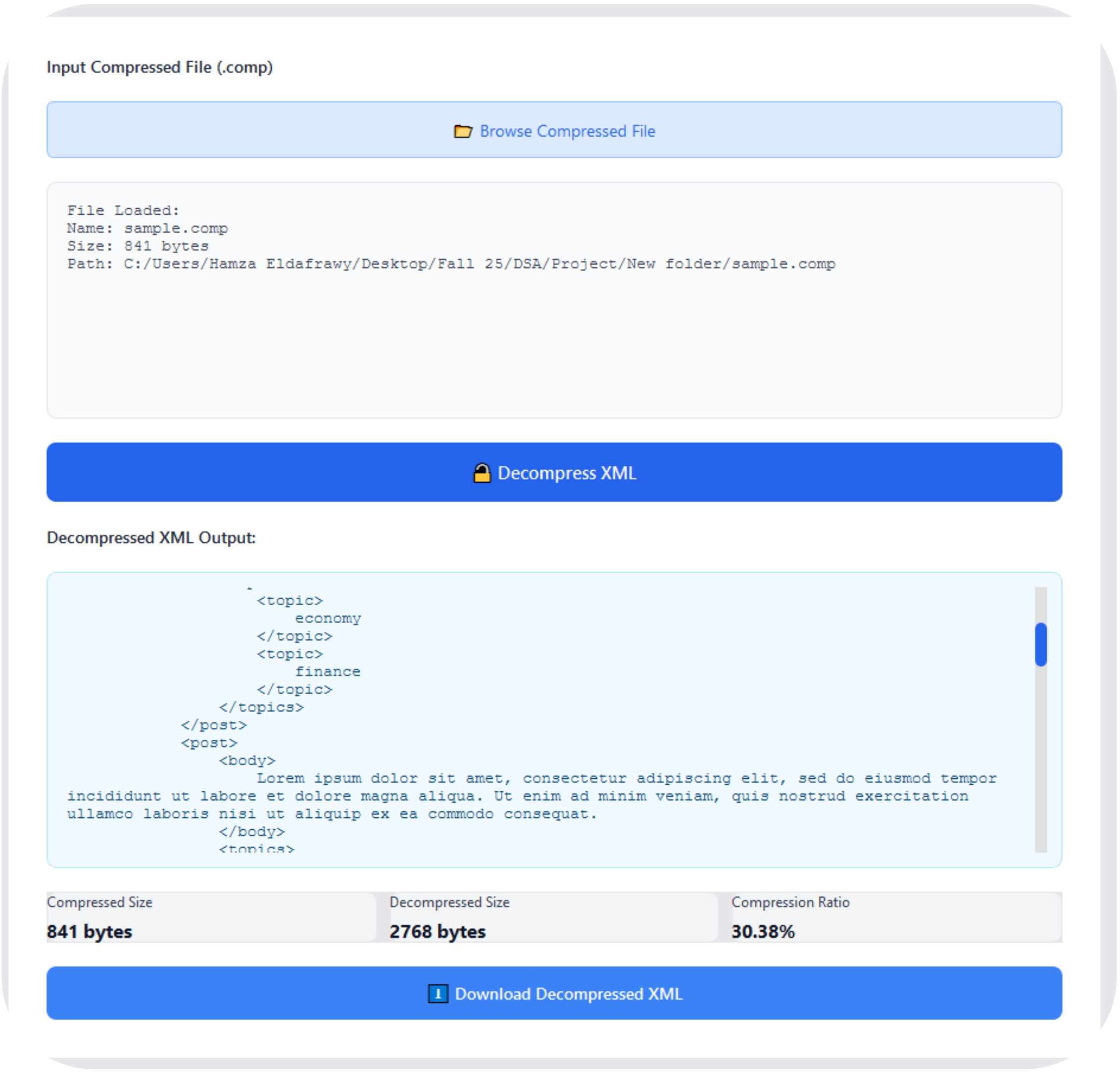
Input XML File

```
<users>
    <user>
        <id>1</id>
        <name>Ahmed Ali</name>
        <posts>
            <post>
                <body>
                    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.
                </body>
                <topics>
                    <topic>
                        economy
                    </topic>
                    <topic>
                        finance
                    </topic>
                </topics>
            </post>
            <post>
                <body>
```

Compressed Output:

Original Size	Compressed Size	Reduction Ratio
2768 bytes	**841 bytes**	**69.62%**

- **Storage and Decompression**: The compressed data is saved as a binary file containing the replacement dictionary and encoded text, allowing for perfect reconstruction of the original XML or JSON file.



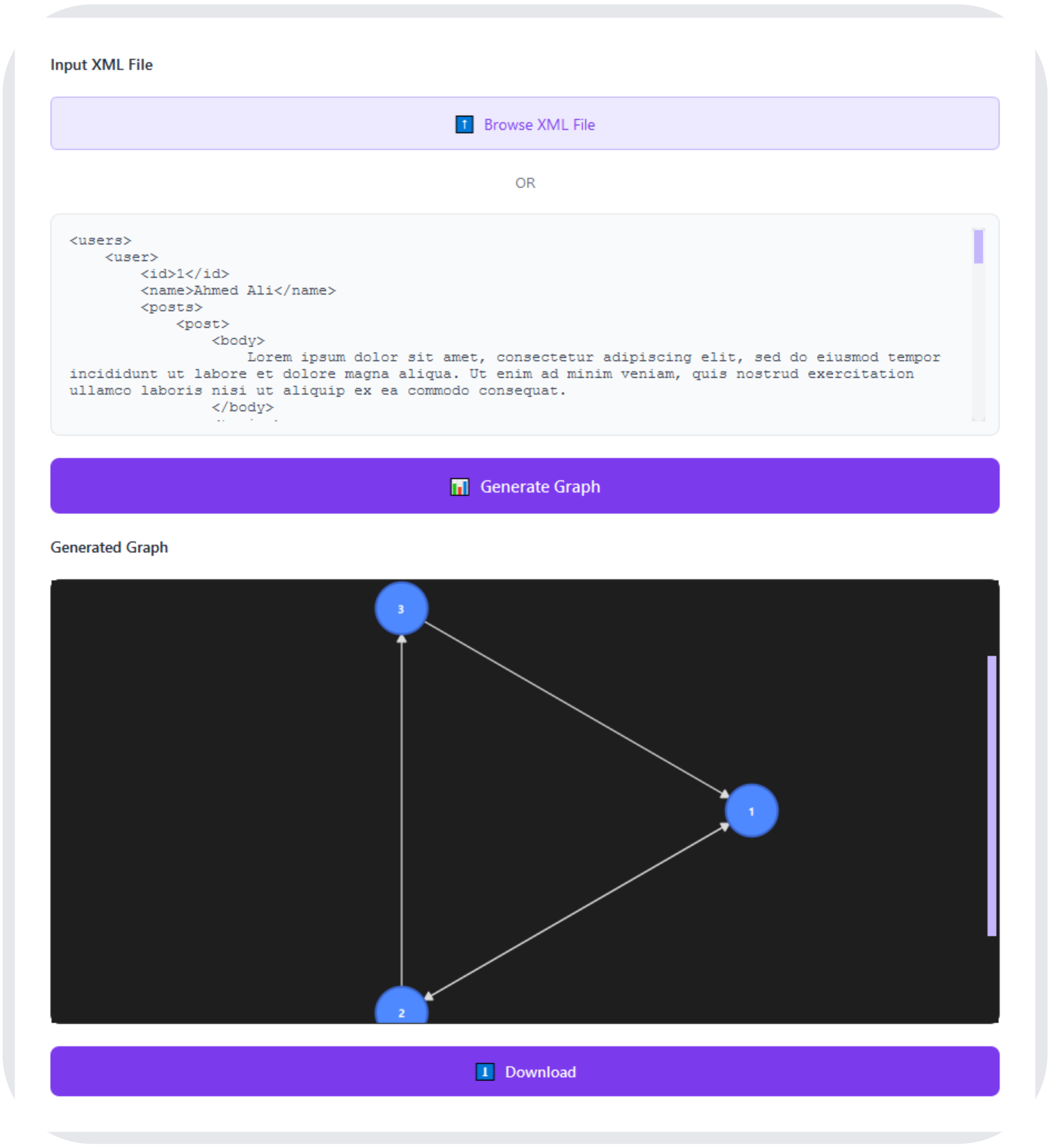
```
File Loaded:
Name: sample.comp
Size: 841 bytes
Path: C:/Users/Hamza Eldafrawy/Desktop/Fall 25/DSA/Project/New folder/sample.comp
```

```
                    <topic>
                        economy
                    </topic>
                    <topic>
                        finance
                    </topic>
                </topics>
            </post>
            <post>
                <body>
                    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.
                </body>
```

Compressed Size	Decompressed Size	Compression Ratio
841 bytes	2768 bytes	30.38%

3.6 Graph Visualization

To fulfill Level 2 requirements, the project integrates with **Graphviz**. The GraphVisualizer converts the adjacency list into the **DOT language**, which is then rendered as an image (e.g., .jpg) showing the directed relations between users in the network.



```
<users>
    <user>
        <id>1</id>
        <name>Ahmed Ali</name>
        <posts>
            <post>
                <body>
                    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.
                </body>
```

3. Complexity of Operations

Efficiency is critical when handling large XML files. Below is the time complexity analysis for the primary operations, where **n** is the number of characters/lines in the file, **u** is the number of users (vertices), **p** is the number of posts users wrote, **m** is the number of length of text inside posts users wrote.

3.1 XML Operations

3.1.1 XML Consistency Check

Description:
Validates XML structure, detects syntax errors, summarizes them, and optionally fixes invalid XML files.

Overall Time Complexity:
O(n), where *n* is the size of the XML input.

Main Functions:

`verify()`	O(n)
`countErrorSummary()`	O(n)
`fixXml()`	O(n)

Helper Functions and Their Complexity:

`readFileToString()`	O(n)
`is_valid_char()`	O(1)
`isValidFirstChar()`	O(1)
`isClosingTag()`	O(1)
`getTagName()`	O(n)
`writeToFile()`	O(n)

3.1.2 XML Minifying

Description:
Removes unnecessary whitespaces and line breaks from an XML file.

Overall Time Complexity:
O(n), where *n* is the number of characters in the file.

Main Functions:

`minifyXML()`	O(n)

3.1.3 XML Prettifying

Description:
Formats XML content with proper indentation and line breaks for readability.

Overall Time Complexity:
O(n)), where *n* is the size of the XML input.

Main Functions:

`prettifyXML()`	O(n)
`tokenizeXML()`	O(n)
`buildTree()`	O(n)
`print_prettified()`	O(n)

3.1.4 Converting XML to JSON

Description:
Transforms XML structure into an equivalent JSON representation.

Overall Time Complexity:
O(n), where *n* is the size of the XML input.

Main Functions:

`convert()`	O(n)	The entry point that initializes the JSON object structure.
`convertToJson()`	O(n)	The core recursive function that traverses the tree and handles logic for objects vs. arrays.
`isLeafWrapper()`	O(1)	Helper check to determine if a node contains only text
`tokenizeXML()`	O(n)	*Pre-requisite:* Parses the raw string into tokens
`buildTree()`	O(n)	*Pre-requisite:* Constructs the Node structure from tokens

---

3.1.5 Compressing XML Files

Description:

Reduces XML file size using frequency-based compression.

Overall Time Complexity:

O(n), where *n* is the size of the XML input.

Main Functions:

`compressXML()`	O(n)
`compress()`	O(n)
`buildFrequencyTable()`	O(n)
`findMostFrequentPair()`	O(n)
`applyPair()`	O(n)
`to_string()`	O(n)

3.1.6 Decompressing XML Files

Description:

Restores a compressed XML file to its original form.

Overall Time Complexity:

O(n), where *n* is the size of the XML input.

Main Functions:

`decompressXML()`	O(n)
`decompress()`	O(n)
`from_string()`	O(n)

3.1.7 XML Parsing (Tree Construction):

Description:

Every tag and text element is visited once to create the node.

Overall Time Complexity:

O(n), where *n* is the size of the XML input.

Main Functions:

`tokenizeLine()`	O(n)
`buildTree()`	O(n)

3.2 Graph Operations

3.2.1 Representing XML as a Graph

Description:

Converts XML data into a graph representation of users and their relationships.

Overall Time Complexity:

Complexity of worst-case scenario, where each user is connected to all other users in the network, is $O(n^2)$ where *n* is the number of users

Main Functions:

`buildGraph()`	$O(u^2)$

Helper Functions and Their Complexities:

`tokenizeXML()`	O(n)
`countUsers()`	O(n)

---

8. Finding the Most Influential User

Description:

Identifies the user with the highest influence based on follower count.

Overall Time Complexity:

O(n), where *n* is the number of users.

Main Functions:

`findMostInfluentialUser()`	O(u)

---

9. Finding the Most Active User

Description:

Determines the user with the highest activity level.

Overall Time Complexity:

O(n^2), where *n* is the number of users.

Main Functions:

`findMostActiveUser()`	**O(u^2)**

---

10. Finding Mutual Followers Between Three Users

Description:
Finds common followers shared between three users.
Overall Time Complexity:
O(n log n)
Main Functions:

`findMutualFollowers()`	**O(u log u)**

Helper Functions and Their Complexities:

`sort()`	**O(u log u)**
`set_intersection()`	**O(n)**

---

11. Post Search

Description:

Searches posts across the entire user network.

Overall Time Complexity:

O(n), where *n* is the total number of posts. Let **k** = number of matches, **m** = text length, **u =** number of users.

Main Functions

`searchPostsByWord()`	O(p*m)
`printMatches()`	O(p)
`extractUserId()`	O(1)
`extractPostText()`	O(1)

Helper Functions and Their Complexities:

`getNeighbors()`	O(log u)
`toLower()`	O(m)
`extractTagContent()`	O(1)

---

12. User Suggestions

Description:

Suggests new users to follow based on network relationships.

Overall Time Complexity:

$O(n^2)$, where *n* is the number of users.

Main Functions:

`suggestUsers()`	$O(u^2)$

Helper Functions and Their Complexities:

`getFollowings()`	$O(u^2)$
`getNeighbors()`	O(log u)
`sort()`	O(u log u)
`erase()`	O(u)
`removeValues()`	O(u)
`unique()`	O(u)

---

4. References

Books

- Data Structure and Algorithms Using C++A Practical Implementation (Sachi Nandan Mohanty, Pabitra Kumar Tripathy)
- Data Structures and Algorithm Analysis in C++ (Mark Allen Weiss)
- Data Structures and Algorithms Made Easy (Narasimha Karumanchi)

Sites

- https://www.w3schools.com/dsa/index.php
  https://www.w3schools.com/dsa/dsa_data_stacks.php
- https://www.w3schools.com/dsa/dsa_theory_hashtables.php
- https://www.w3schools.com/dsa/dsa_theory_trees.php
- https://www.w3schools.com/dsa/dsa_theory_graphs.php
- https://www.geeksforgeeks.org/dsa/dsa-tutorial-learn-data-structures-and-algorithms/
- https://www.geeksforgeeks.org/dsa/graph-data-structure-and-algorithms/
- https://doc.qt.io/all-topics.html
- https://doc.qt.io/qt-6/qtwidgets-index.html

4.1 Project Repository

Source Code & Documentation: GitHub

Install file

https://github.com/amr10w/XML-Social-Network-Analyzer/blob/main/installation/ilovexml_installer.exe