Project Report: XML Parser and Social Network Visualizer

Course: CSE331 - Data Structures and Algorithms

Committed by: Dr.Islam Ahmed Mohamoud Elmadah ,Eng. Fady Fargallah

Name	ID	Role
Mohamed Hamada Hassan	2200820	SocialNetwork Part 2 & Report
Amr Ashraf Hussien	2201048	Social Network Part1 & Gui
Yousef Ahmed Mohammed	2200405	Graph Architect
Eyad Tarek Nagy	2200512	Xml Pretfier
Abdulla Mohamed	2200423	GUI & Integration
Mohamed Ehab	2201067	Xml Validator
Ahmed Mahmoud Elmorsy	2200725	Report
Seif Mohamed	2200929	CLI Implementation
Mohamed Walid	2200243	Xml To Json Converter
Mahmoud Shaban	2200676	XML Minfier & Compressor & Decompressor

Abstract

This report details the design and implementation of a multifaceted software tool developed for the CSE331 Data Structures and Algorithms course. The project, an "XML Parser and Social Network Visualizer," provides a comprehensive solution for processing, analyzing, and visualizing social network data represented in XML format. The system operates in two modes: a command-line interface (CLI) for scriptable operations and a graphical user interface (GUI) for interactive use. Core functionalities include XML validation using a custom stack-based implementation, document formatting, minification, conversion to JSON, and data compression/decompression using the DEFLATE algorithm. A key component of the project is the from-scratch implementation of a graph data structure (using an adjacency list) to model the social network, enabling advanced analysis such as identifying influential users, finding mutual connections, and suggesting new followers. The report covers the theoretical background, architectural design, implementation details of core data structures and algorithms, and a thorough analysis of their computational complexity.

Table Of Contents

Background

In the modern digital landscape, data is generated and exchanged at an unprecedented scale. Two fundamental concepts underpin much of this activity: structured data representation and the analysis of interconnected relationships. This project sits at the intersection of these two domains, leveraging Extensible Markup Language (XML) as a data format and graph theory as an analytical framework to build a tool for social network analysis.

Introduction to XML

XML, or Extensible Markup Language, is a markup language designed to store and transport data. Unlike HTML, which is designed to display data with predefined tags, XML allows users to define their own self-descriptive tags, making it a flexible and powerful tool for data organization. An XML document is composed of a hierarchy of elements, each delimited by a start-tag and an end-tag. This hierarchical structure naturally forms a tree, where a single root element contains all other elements as its children or descendants. This tree-like structure is fundamental to how XML is processed and manipulated by software.

A software module known as an XML processor, or parser, is used to read an XML document and provide an application with access to its content and structure. Parsers can validate a document's correctness at two levels: well-formedness (adherence to XML syntax rules) and validity (conformance to a schema like a Document Type Definition (DTD) or XML Schema Definition (XSD)).

Social Networks as Graph Data Structures

A social network, at its core, is a collection of entities (e.g., people, organizations) and the relationships that connect them. This structure is perfectly modeled by a mathematical graph, a data structure consisting of nodes (or vertices) and edges (or ties). In the context of a social network, each user can be represented as a node, and a relationship, such as "follows" or "friend of," can be represented as an edge connecting two nodes.

By representing a social network as a graph, we can apply a rich set of algorithms to uncover valuable insights. Social Network Analysis (SNA) is the field dedicated to this pursuit, using graph-based metrics to identify key players, detect communities, and understand the flow of information. Common metrics include centrality measures (degree, betweenness, closeness) which help quantify the importance of a node within the network.

Project Objectives

This project aims to develop a robust desktop application that bridges the gap between raw, XML-formatted social network data and actionable insights derived from graph analysis. The primary objectives are:

1. To implement a tool capable of parsing and validating large XML files representing a social network of users, posts, and followers.
2. To provide essential XML manipulation utilities, including consistency checking, error correction, formatting, minification, and conversion to the popular JSON format.
3. To implement a data compression and decompression mechanism to efficiently store and transmit XML data.
4. To construct a graph data structure from scratch to model the social network's user-follower relationships.
5. To implement various network analysis algorithms to extract meaningful social metrics, such as identifying the most influential and active users.
6. To offer both a command-line interface (CLI) for automation and a user-friendly graphical user interface (GUI) for interactive analysis and visualization.

Implementation Details

The system was developed in C++, leveraging its extensive standard library and rich ecosystem of third-party packages for GUI development and data handling. The implementation is divided into logical modules corresponding to the project's functional requirements.

Core Data Structures

As per the project guidelines, fundamental data structures like arrays and linked lists were used from standard libraries, while more complex structures central to the project's logic were implemented from scratch.

Stack for XML Validation

To check for XML consistency (i.e., well-formedness), a **Stack** data structure was implemented. A stack operates on a Last-In, First-Out (LIFO) principle, with its primary operation (push, pop, and `peek)` .

The validation algorithm works by parsing the XML file token by token. When an opening tag (e.g., `<user>`) is encountered, its name is pushed onto the stack. When a closing tag (e.g., `</user>`) is found, the stack is checked. If the stack is empty or the tag at the top of the stack does not match the closing tag, an error is reported. If they match, the tag is popped from the stack. At the end of the document, if the stack is empty, the document is consistent; otherwise, it means there are unclosed tags.

Graph for Social Network

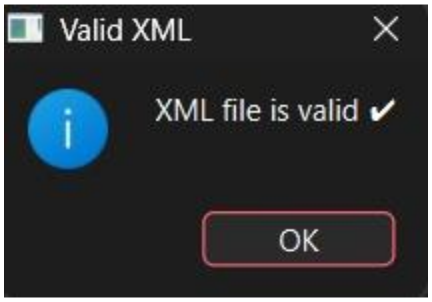
The core of the social network analysis functionality relies on a custom-built **Graph** data structure. Given that social networks are typically sparse, an **Adjacency List** representation was chosen over an Adjacency Matrix. An adjacency list uses O(V + E) space, where V is the number of vertices (users) and E is the number of edges (follower relationships), which is significantly more efficient than the O(V2) space required by an adjacency matrix.

Our implementation consists of a `Graph` class that manages a dictionary. The keys of this dictionary are the unique user IDs, and the values are `Vertex` objects. Each `Vertex` object stores user data (name, posts) and maintains a list of adjacent vertices (followers and following).

Level 1 Features: XML Processing

XML Parsing, Validation, and Correction

The `verify` operation first uses the custom stack-based algorithm to check for well-formedness. It reads the file, pushing opening tags and popping matching closing tags. Errors, along with their line numbers, are logged. If the `-f` (fix) option is enabled, the system attempts to correct errors. Simple errors like a mismatched closing tag are corrected by replacing it with the tag expected from the stack's top. Missing closing tags at the end of the file are added by popping remaining tags from the stack and appending them.


Valid XML

XML file is valid ✔

XML Formatting (Prettifying) and Minification

- I make function takes two file paths:
    - `inputFilePath` → path of the original XML file
    - `outputFilePath` → path of the formatted XML file
- It opens the input file to read the XML content.
- It opens the output file to write the formatted XML.
- The entire XML file is read line by line and stored in a single string called **`xml`**.
- An index `i` is used to scan the XML string **character by character**.
- An `indentLevel` variable is used to track the current indentation level.
- When the character < is found:
    - A complete XML tag is read until > is reached.
    - The helper function **`isClosingTag()`** is used to check if the tag is a closing tag.
    - If it is a closing tag, the indentation level is decreased before writing.
    - The helper function **`indent()`** is used to generate proper indentation.
    - The tag is written to the output file with correct indentation.
    - If the tag is an opening tag, the indentation level is increased after writing.
- When text between tags is found:
    - The text is read until the next < character.
    - The helper function **`trimAll()`** is used to remove extra spaces and normalize whitespace.
    - Empty or whitespace-only text is ignored.
    - Valid text is written to the output file with proper indentation.
- The process continues until all characters in the XML string are processed.
- Finally, the input and output files are closed.

Time And Space complexity:

- Format_XML_File(const string& inputFilePath, const string& outputFilePath)
- Reads XML from a file path then put it in a String and Formats it then Writes formatted XML to an output file.
- It also uses helper functions like:
    - **1- isClosingTag()**:
        - Checks if a tag is a closing tag like </user>.
        - Time Complexity:O(1) ,SpaceComplexity: O(1)

- **2- indent():**
    - Creates a string of spaces for indentation.
    - Time Complexity:O(1) ,SpaceComplexity: O(1)
- **3- trimAll()**
    - Iterates once over the text and removes extra spaces/tabs/newlines
    - Time Complexity:O(n) where n : text length ,SpaceComplexity: O(n)

Finally complexity:

- Time Complexity: O(n)+O(n)+O(n)+O(1)+O(1) = **O(n)**
- Space Complexity: O(n)+O(n)+O(1)+O(1) = **O(n)**

XML to JSON Conversion

Core Components and Logic

A. Pre-processing (`trimAll`)

Before parsing, the `trimAll` function cleans the input string. It removes all whitespace, tabs, and newlines. This ensures the parser doesn't encounter unexpected "invisible" characters while checking for the next < or > symbols.

B. Recursive Parsing (`parseXML`, `readTag`, `readText`)

The heart of the converter lies in its recursive logic:

- **`readTag`**: Extracts the name of the XML tag (e.g., converts `<name>` to "name").
- **`readText`**: Captures the string value located between an opening and closing tag.
- **`parseXML` (Recursive)**:
    - It identifies an opening tag and maps it to a JSON "key".
    - If it encounters another opening tag immediately after, it calls itself (**recursion**) to create a nested JSON object.
    - If it encounters plain text, it treats it as a JSON "value".
    - It handles comma placement between key-value pairs using a `first` boolean flag.

C. Post-processing (`format`)

The raw JSON generated by the parser is a single, dense line. The `format` function:

- Tracks indentation levels.
- Adds newlines after braces `{}` and commas `,`.
- Ensures that colons `:` are followed by a space for better readability.
- Includes a safety check (`inString`) to ensure characters inside quotes are not incorrectly formatted.

3. Data Flow Execution

The `convert` function manages the end-to-end lifecycle of the process:

1. **File I/O**: Reads the XML source file into a string buffer.
2. **Transformation**: Calls `trimAll` -> `parseXML` -> `format`.
3. **Persistence**: Writes the final, prettified string to `data.json`.

4. Summary of Key Functions

Function	Responsibility
`trimAll`	Removes all whitespace for a "clean" parse.
`readTag`	Moves the pointer and returns the tag name.
`parseXML`	Logic engine that handles nesting and JSON structure.
`format`	Adds tabs, newlines, and spacing to the final string.
`convert`	Main interface for file handling and pipeline execution.

4. Summary of Key Functions

Time Complexity: O(N)

Space Complexity: O(N)

Compression and Decompression

For the compress and decompress operations, we implemented a solution based on the **DEFLATE** algorithm using a C++ compression library such as **zlib**. DEFLATE is a widely used lossless compression algorithm that combines the **LZ77** algorithm with **Huffman coding**. During compression, the raw byte data from the input file is passed to the zlib compression functions (e.g., compress()), producing a compressed byte stream that is written to the output .comp file. The decompression process uses the corresponding zlib functions (e.g., uncompress()) to restore the original data from the compressed file.

Core Components and Logic for compression:

High-level idea

This code implements a **pair-substitution compression algorithm** (very similar to **Byte Pair Encoding – BPE**).
It repeatedly finds the **most frequent adjacent character pair**, replaces it with a **new unused character (token)**, records this mapping in a dictionary, and updates the sequence until no useful compression is possible.

Main data structures

- **map<string,int> pairCount**
  Counts frequencies of all adjacent 2-character pairs.
- **map<char,bool> charexist**
  Tracks which ASCII characters already appear in the data.
- vector<char> unusedChar
  Stores characters not used in the input, later used as new tokens.
- dictionary_token / dictionary_pair
  Store compression rules:
  token → pair and pair → token.
- priority_queue<Pair>
  Max-heap ordered by pair frequency (most frequent pair on top).

Function-by-function explanation

initMaps(string &data_sequence)

1. Initializes all ASCII characters as unused.
2. Scans the input sequence once:
    a. Marks seen characters.
    b. Counts all adjacent pairs.

3. Pushes pairs with frequency > 1 into the priority queue.
4. Collects unused characters to serve as compression tokens.

Purpose: Prepare frequency statistics and available tokens.

update_pair_counts_and_pq(const string &data_sequence)

1. Clears previous pair counts and priority queue.
2. Recomputes adjacent pair frequencies for the updated sequence.
3. Rebuilds the priority queue with pairs occurring more than once.

Purpose: Refresh statistics after each replacement step.

merge_and_update_sequence(string &sequence, string oldPair, char newToken)

- Finds all occurrences of oldPair.
- Replaces each occurrence with the single-character newToken.

Purpose: Perform the actual compression step.

compress(inputfileName, outFileName)

1. Reads the entire input file into memory.
2. Initializes maps and priority queue.
3. Repeatedly:
   a. Extracts the most frequent pair.
   b. Assigns a new unused character as its token.
   c. Replaces the pair in the sequence.
   d. Recomputes pair statistics.
4. Writes:
   a. The compression dictionary.
   b. A marker (###).
   c. The compressed sequence.

Purpose: Full compression pipeline.

Time complexity

Let:

- **n** = length of the input sequence
- **k** = number of compression iterations (bounded by available unused characters, ≤ 255)

Per iteration:

- Replacing pairs: O(n)
- Recounting pairs: O(n)
- Priority queue rebuild: O(n log n) in worst case

Total: O(nlogn)

Space complexity

- Pair counts: O(n)
- Priority queue: O(n)
- Dictionaries and auxiliary maps: O(n)
- Stored input sequence: O(n)

Core Components and Logic for decompression:

High-level idea

This code implements the **decompression phase** of a pair-substitution compression algorithm (similar to Byte Pair Encoding – BPE).
It **reconstructs the dictionary** stored in the compressed file, then **reverses all token substitutions** in the correct order to recover the original data sequence.

Main data structures

map<char, string> dictionary_token_decomp
Stores decompression rules:
token → original 2-character pair.

string data_sequence
Holds the compressed data after removing the dictionary header.

Function-by-function explanation

decompress(string inputfileName, string outFileName)

Reads the entire compressed file into memory.

Reconstructs the dictionary:

- Reads entries of the form:
    - one character token
    - followed by its original two-character pair
- Stops when the marker "###" is encountered.
- Removes dictionary data from the input string.
- Stores mappings in dictionary_token_decomp.

Expands tokens back to original pairs:

- Iterates through the dictionary **in reverse order**.
- For each token:
    - Finds all occurrences in the compressed sequence.
    - Replaces each token with its original pair.

Writes the fully decompressed sequence to the output file.

Purpose: Fully reverse the compression process and restore the original data.

Time complexity

Let:

- **n** = length of the compressed data sequence
- **k** = number of dictionary entries (tokens)

Dictionary extraction:

- Each dictionary entry is read once O(k)

Decompression phase:

- For each token:
    - Full scan and replacement of the sequence: O(n)

Total: O(k*n)

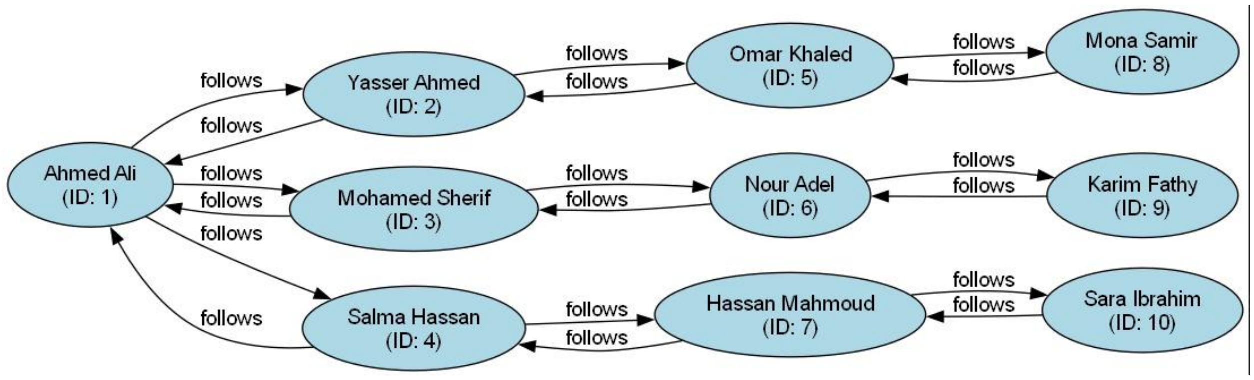
Space complexity

- Compressed data storage: O(n)
- Decompression dictionary: O(k)
- Temporary variables: O(1)

Level 2 Features: Social Network Analysis

Graph Construction and Visualization

After parsing the input XML, the system iterates through each element. For each user, `addUser()` is called on our custom `Graph` object. Then, for each follower listed within a user's tag, `addFollower()` is called to create a directed edge from the follower to the user being followed. Posts and topics are stored within their respective `User` objects.



Network Analysis Algorithms

The analysis functions operate directly on our custom graph data structure:

- **Most Influential User:** This is determined by finding the user with the highest number of followers (in-degree). The algorithm iterates through all vertices in the graph, keeps track of the vertex with the maximum followers, and returns that user's ID and name. This is a measure of Degree Centrality.
- **Most Active User:** "Activity" is defined as the total number of connections following, or total degree. A similar iteration is performed to find the user with the maximum following.
- **Mutual Followers:** For a given list of user IDs, the algorithm retrieves the set of followers for each user. It then computes the intersection of these sets to find the followers common to all specified users.
- **Follower Suggestions:** For a given user, this function implements a 2-Level traversal. It first gets the set of users that the target user is already following. Then, it iterates through each of those followed users (the "friends") and collects their followers ("friends of friends"). Finally, it returns the collected users after removing those the target user already follows and the user themselves.

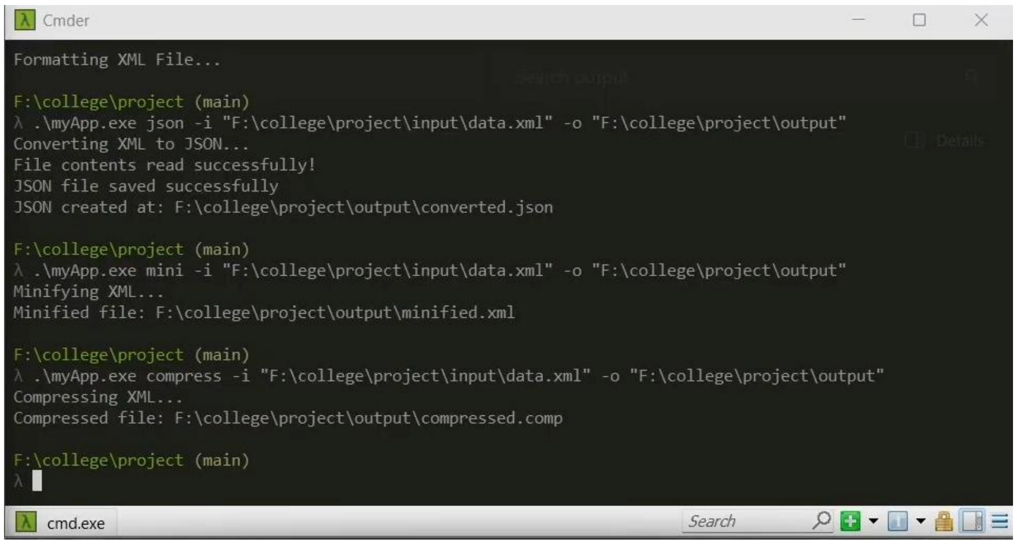
Post Search

To enable efficient post searching, an **inverted index** was implemented using C++ standard library containers. During the initial parsing phase, as posts are added to user vertices, each post’s body and topics are tokenized into individual words. The inverted index maps each word or topic to a collection of entries containing the corresponding user_id and a reference (or pointer) to the associated post object. This structure is typically implemented using unordered_map, allowing average-case lookup time. When a search query is performed, the system directly retrieves the list of all posts associated with the searched term, enabling fast and efficient query processing.

User Interface Design

Command-Line Interface (CLI)

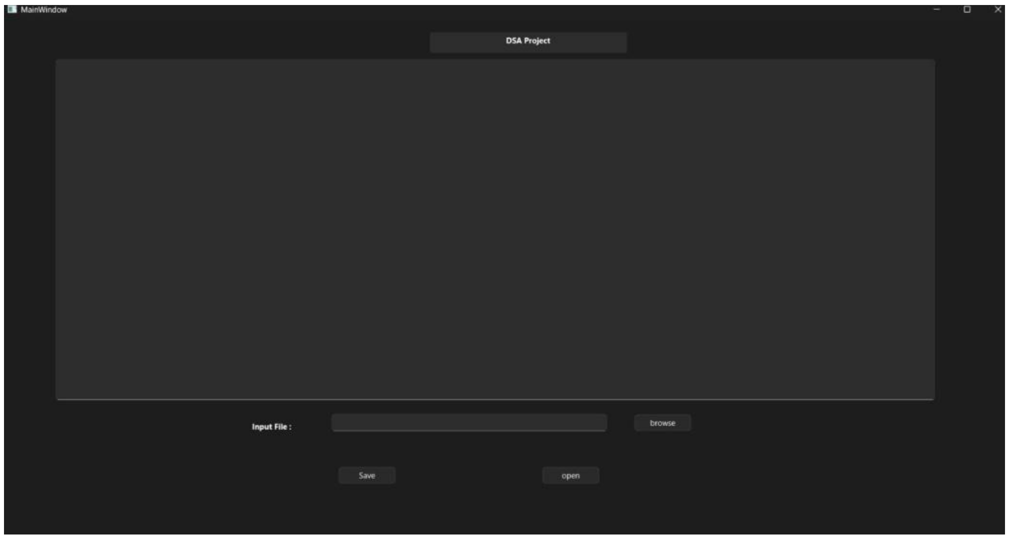
The command-line interface (CLI) was implemented using C++ command-line argument parsing techniques and libraries (such as getopt, getopt_long). It provides a structured way to define commands (e.g., verify, format), positional arguments, and optional flags (such as -i for input and -o for output). This design enables clear command syntax, robust argument validation, and seamless integration of the tool into automated scripts and workflows.



```
Formatting XML File...

F:\college\project (main)
λ .\myApp.exe json -i "F:\college\project\input\data.xml" -o "F:\college\project\output"
Converting XML to JSON...
File contents read successfully!
JSON file saved successfully
JSON created at: F:\college\project\output\converted.json

F:\college\project (main)
λ .\myApp.exe mini -i "F:\college\project\input\data.xml" -o "F:\college\project\output"
Minifying XML...
Minified file: F:\college\project\output\minified.xml

F:\college\project (main)
λ .\myApp.exe compress -i "F:\college\project\input\data.xml" -o "F:\college\project\output"
Compressing XML...
Compressed file: F:\college\project\output\compressed.comp

F:\college\project (main)
λ
```

Graphical User Interface (GUI)

The graphical user interface (GUI) was developed using the Qt framework in C++. A QMainWindow class was used as the main application window, providing a standard layout that includes menus and a status bar. The central area of the application consists of several key widgets organized using Qt layout managers such as QVBoxLayout and QFormLayout.



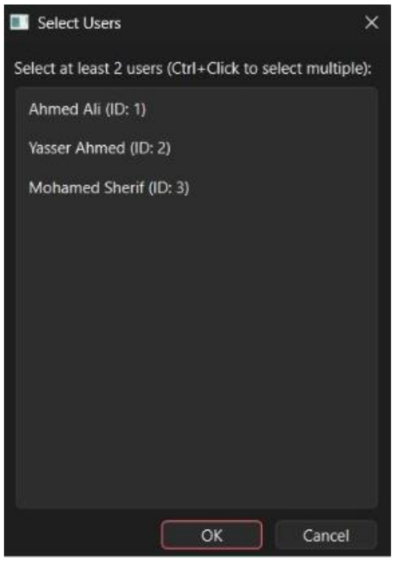
File Input:
A QLineEdit is used to display the selected file path, accompanied by a QPushButton that opens a QFileDialog to allow users to browse and select files.

Operation Buttons:
A set of QPushButton widgets is provided, with one button corresponding to each supported operation (e.g., Verify, Format, Minify).

Text Areas:
Two large, read-only QTextEdit widgets are used to display the input XML content and the output produced by the selected operation.

Status Bar:
The QStatusBar is used to provide real-time feedback to the user, such as “File loaded successfully” or “Validation complete: 2 errors found.”

Each button’s clicked signal is connected to a corresponding **slot** function, following Qt’s signal–slot mechanism. These slot functions invoke the appropriate backend logic and update the output text area with the resulting data.


Select Users

Select at least 2 users (Ctrl+Click to select multiple):

Ahmed Ali (ID: 1)

Yasser Ahmed (ID: 2)

Mohamed Sherif (ID: 3)

3. Complexity of Operations

The efficiency of the implemented algorithms is critical, especially when dealing with large XML files and complex social networks. The time and space complexity for key operations are analyzed below, where *N* is the number of nodes/elements in the XML tree, *V* is the number of users (vertices), and *E* is the number of follower relationships (edges).

3.1 Time Complexity

The time complexity measures how the execution time of an algorithm scales with the input size.

Operation	Time Complexity	Justification
XML Validation (Stack)	O(N)	The parser makes a single pass through the XML document, performing a constant number of stack operations for each tag.
DOM Tree Construction	O(N)	Standard DOM parsers typically build the tree in linear time relative to the number of elements and text nodes.
XML Formatting/Minifying	O(N)	Requires a full traversal of the in-memory DOM tree to serialize the output.
XML to JSON Conversion	O(N)	Requires a full traversal of the DOM tree to build the dictionary representation.
Graph Construction	O(V + E)	Involves iterating through all users and their follower lists once to add vertices and edges.
Find Most Influential/Active User	O(V)	Requires iterating through all vertices once to find the maximum degree.
Find Mutual Followers (k users)	O(k * F_max)	Where F_max is the maximum number of followers for any of the k users. Involves set intersection operations.
Suggest Followers (for user u)	O(deg(u) * deg_avg_f)	Where deg(u) is the number of users u follows, and deg_avg_f is the average number of followers of those users. Involves a 2-hop neighborhood traversal.
Post Search by word (Inverted Index)	O(P * L) on average	Where L is the avg post length and P is the total number of posts

3.2 Space Complexity

Space complexity measures the amount of memory an algorithm requires in relation to the input size.

Operation	Space Complexity	Justification
XML Validation (Stack)	O(D)	The stack's maximum size is determined by the maximum nesting depth (D) of the XML tree.
DOM Tree Construction	O(N)	The entire XML document is loaded into memory as a tree of objects.
Graph Representation (Adjacency List)	O(V + E)	Space is required to store all vertices and all edges (follower relationships) [29].
Compression (zlib)	O(N)	The \`zlib\` library requires holding chunks of the input data in memory during compression.
Post Search (Inverted Index)	O(P)	The inverted index stores every unique word (P) and a list of pointers to the posts containing it.

Github Repo &Video Demo

- https://github.com/amrra4rf/XML-Network-Editor
- https://drive.google.com/file/d/1JkDmq8WRpXIuHHwoyHc_tOX3n-_P0-Ri/view?usp=sharing

References

- [1] W3Schools, "XML Tutorial," [Online]. Available: https://www.w3schools.com/xml/.
- [2] M. P. J. Team, "XML Tutorial," University of Maryland, 2017. [Online]. Available: https://terpconnect.umd.edu/~austin/ence688r.d/lecture-handouts/XML-Tutorial2017.pdf.
- [3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," W3C, 26-Nov-2008. [Online]. Available: https://www.w3.org/TR/xml/.

- [4] C. Li, "XML Parsing," Encyclopedia of Database Systems, 2009. [Online]. Available: https://ranger.uta.edu/~cli/pubs/2009/XMLParsing_ChengkaiLi.pdf.
- [5] H2K Infosys, "How to Validate XML using XSD and DTD," 21-Nov-2020. [Online]. Available: https://www.h2kinfosys.com/blog/how-to-validate-xml-using-xsd-dtd/.
- [6] A. Navlani, "Social Network Analysis in Python," DataCamp, 02-Oct-2018. [Online]. Available: https://www.datacamp.com/tutorial/social-network-analysis-python.
- [7] GeeksforGeeks, "Introduction to Social Networks using NetworkX in Python," 15-Jul-2025. [Online]. Available: https://www.geeksforgeeks.org/machine-learning/introduction-to-social-networks-using-networkx-in-python/.