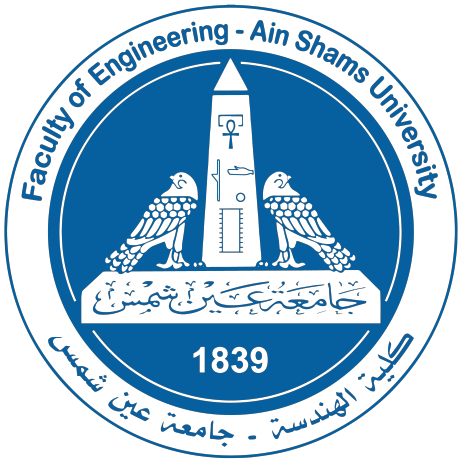

Data Structures and Algorithms [CSE331s]

XML Visualiser Project

Name	ID
Mina Nasser Shehata	2100370
Mostafa Hamada Hassan Ahmed	2100587
Zeyad Magdy Roushdy	2100393
Ahmed Ashraf Ahmed	2100476
Mina Antony Samy Fahmy	2101022

Contents

List of Figures

1 Background

This project is an XML Visualiser/Editor. We will begin by looking at its features.

1.1 Features

1. XML Operations
   (a) XML Consistency Verification
   (b) Error fixing : Fixes consistency errors
   (c) Formatting : Pretty printing
   (d) Converting XML to JSON
   (e) Minifying the XML file : Minimizes the size of the XML file through deleting the whitespaces and indentations
   (f) Compression/Decompression
2. Graph Operations These operations are are for the purposes of representing user data in a social network (i.e : followers, posts, activity, etc...):
   (a) Graph drawing : Outputs an image representing the users' names and how they are connected together (through following)
   (b) Finding the most active user (the one connected to the most number of users)
   (c) Finding the most influential user (the one with the most followers)
   (d) Finding mutual followers between a given set of users
   (e) Suggest for a user a set of users to follow (the followers of his followers)
   (f) Searching through posts : using a search word or topic, find the post where said search term or topic was mentioned

1.2 Operation Modes

The program can operate in 2 modes:

- Graphical User Interface (GUI)
- Command Line Interface (CLI)

1.2.1 GUI mode

We can see the following in Figure 1.1

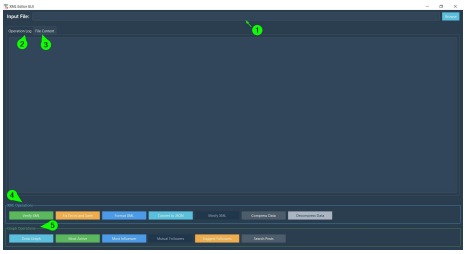

Figure 1.1: GUI

1. Input file field, Usage:
   Click “Browse” and choose the file through the file explorer pop-up
2. Operation log : a read-only text field where the log/result of the last used command appears
3. File Content : Displays content of file after importing it into the program, or the results of using the “XML operations” mentioned in Section 1.1
4. The Set of XML operations
   - All of these operations will show a file explorer pop-up to save the contents of the operation except for “Verify XML”
5. The Set of Graph operations
   - “Draw Graph” will show a file explorer pop-up to save the graph image and other operations will display their output in the operation log

1.2.2 CLI Mode

The program has the following format in CLI mode:

CLI General Format

```
xml_editor [OPTIONS] [-w]{keyword} [-t]{topic} -i input_file.[xml/comp] [-f]  [
-id]{ID} [-ids]{IDs } [-o output_file.[xml/json/jpg/comp]]
```

We will explore it through examples

(1) Checking the XML consistency and fixing

```
xml_editor verify -i input_file.xml [-f -o output_file.xml]
```

- **`verify`** is used to output if the xml file is valid or not, the erroneous line numbers and how many errors there are in the input file.

- The part **[-f -o output_file.xml]** is optional, it's used to fix the xml file if it contains errors and outputs the corrected result into **output_file.xml**

(2) Format/Prettify the XML file

```
xml_editor format -i input_file.xml -o output_file.xml
```

Formats **input_file.xml** and outputs the result into **output_file.xml**

(3) XML to JSON Conversion

```
xml_editor json -i input_file.xml -o output_file.json
```

Converts **input_file.xml** to **output_file.json**

(4) Minify the XML file

```
xml_editor mini -i input_file.xml -o output_file.xml
```

Minifies **input_file.xml** and outputs the result into **output_file.xml**

(5) Compress XML file

```
xml_editor compress -i input_file.xml -o output_file.comp
```

Compresses **input_file.xml** into **output_file.comp**

(6) Decompress XML file

```
xml_editor decompress -i input_file.comp -o output_file.xml
```

Decompresses **input_file.comp** into **output_file.xml**

(7) Draw XML file as Graph

```
xml_editor draw -i input_file.xml -o output_file.jpg
```

Draws **input_file.xml** as a graph in **output_file.jpg**

(8) Find Most active user

```
xml_editor most_active -i input_file.xml
```

Outputs the most active user's name and ID

(9) Find Most Influential User

```
xml_editor most_influencer -i input_file.xml
```

Outputs the most influential user's name and ID

(10) Find Mutual followers between multiple users

```
xml_editor mutual -i input_file.xml -ids id1,id2,id3
```

Find the mutual followers between the users with the following IDs: **id1**, **id2**, **id3**

(11) Suggest a list of users to follow for a certain user (the followers of his followers)

```
xml_editor suggest -i input_file.xml -id id
```

Suggest a list of users to follow for the user with ID : **id**. The suggested users are the followers of his followers

(12) Search posts for a certain keyword

```
xml_editor search -w word -i input_file.xml
```

Outputs posts that contain the keyword "**word**"

(13) Search posts for a certain topic

```
xml_editor search -t topic -i input_file.xml
```

Outputs posts that contain the topic "**topic**"

1.3 Github Repository

Checkout out the source code on our Github repository:
https://github.com/zyadmagdy11/XMLVisualizer

1.4 Test Recordings

You can find the program under testing in these recordings:

- GUI Test CLI Test

2 Implementation Details

Here we will delve into the details of how our project works.
The project is logically divided into two levels.

2.1 Level (1)

2.1.1 XML Consistency Check and Error Fixing

This is considering the file `XML_Consistency.cpp`

2.1.1.1 `bool checkXMLConsistency(string xml)`

This code checks if an XML string is well-formed and consistent by ensuring that all opening tags have corresponding closing tags in the correct order. It uses a stack to keep track of open tags and verifies if each closing tag matches the most recent open tag. If any mismatch or unbalanced tags are found, it returns `false`; otherwise, it returns `true`.
Time Complexity : $O(n)$
Space Complexity : $O(k)$
Where n is the length of the XML string and k is the maximum number of nested tags

2.1.1.2 `vector<int> findMismatchedTags(string xml)`

This function, identifies mismatched XML tags in a given string. It checks for two types of mismatches:

- Unmatched Closing Tags: A closing tag appears but no corresponding opening tag exists in the stack.
- Unmatched Opening Tags: Opening tags that were not closed by the end of the string.

The function performs the following steps:

- Scans the string and identifies tags enclosed in `<` and `>`.
- For opening tags:
  - Adds the tag name to `tagStack`.
  - Adds the tag's position to `positionStack`.
- For closing tags:
  - Looks for a matching opening tag in the stack.
  - If a match is found, removes the matching tag from `tagStack` and `positionStack`.

  - If no match is found, records the position of the mismatched closing tag in `mismatchedPositions`.

- After processing the entire string:
  - Adds any unmatched opening tags (from `positionStack`) to `mismatchedPositions`.
  - Sorts `mismatchedPositions` using `heapSort`.

Returns a sorted vector of all mismatched tag positions.

Time Complexity:

- Processing the XML string:
  - Each character is processed once.
  - Finding the position of $>$ with `find` takes $O(n)$ in the worst case.
  - Combined traversal of the string is $O(n)$
- Matching Closing Tags:
  - For each closing tag, the algorithm may search through the `tagStack`, which takes $O(k)$ where k is the number of tags in the stack.
  - In the worst case, every tag is checked, so this adds up to $O(n^2)$ if all tags are unmatched.
- Inserting Unmatched Tags:
  - Appending unmatched tags to `mismatchedPositions` is $O(n)$
- Sorting Mismatched Positions:
  - `heapSort` on n positions takes $O(n \log(n))$

  Total Time Complexity : $O(n^2)$ (due to stack traversal during mismatched closing tags).

Space Complexity:

- Tag Stack and Position Stack: Store up to n tags and their positions $O(n)$
- Mismatched Positions: Stores up to n positions $O(n)$
- Auxiliary Space for heapSort: $O(1)$
  Total Space Complexity: $O(n)$

2.1.1.3 `string correctMismatchedTags(string xml, vector<int> tag_index)`

This function is designed to fix mismatched or missing XML tags in a given string of XML:

- **Input Parameters:**

  - `xml`: A string representing the XML content.
  - `tag_index`: A vector of indices where mismatched tags are found.
- **Process:**
  - The function first identifies whether an opening or closing tag is missing, using the indices in `tag_index` to locate the mismatched tags.
  - It builds a list of tags (`tagContent`), and for each mismatched tag:
    * If an opening tag is missing a closing tag, it inserts the corresponding closing tag after it.
    * If a closing tag is missing an opening tag, it inserts the corresponding opening tag before it.
  - It adjusts the string's position using `offset` to account for the change in length as tags are added.
- **Output:**
  - The function returns the corrected XML string with the missing or mismatched tags fixed.

Time Complexity:

- **Tag Processing:** Each mismatch is handled one by one. For each index in `tag_index`, the function performs operations like `xml.find()`, `xml.substr()`, and `xml.insert()`. The time complexity of these operations depends on the length of the string being processed. Specifically:
  - `xml.find()` and `xml.substr()` both take $O(n)$ in the worst case, where n is the length of the string.
  - `xml.insert()` also takes $O(n)$ because it might need to shift characters to accommodate the new tag.
- **Overall Complexity:**
  - If there are m mismatched tags, the function processes each tag in $O(n)$ time (with n being the length of the string). Therefore, the time complexity is $O(n \times m)$.

Space Complexity:

- **String Storage:** The function uses additional space to store the corrected `xml` string, which could be as large as the original string plus the tags being inserted. Thus, the space complexity is $O(n)$, where n is the length of the string.
- **Tag List:** The function also uses a `vector<string>` to store tag names, but this list has at most m elements, where m is the number of mismatched tags. This contributes $O(m)$ space.

- **Overall Space Complexity:** $O(n + m)$

Here, m is the number of mismatched tags, and n is the length of the `xml` string.

2.1.2 XML Formatting (Prettifying)

This is an explanation of file `Formatting.cpp` and header `Formatting.h`

2.1.2.1 Initialize Variables

- `output`: Stores the formatted XML result.
- `tagStack`: Keeps track of opened tags for indentation.
- `currentTag`: Temporarily stores the tag currently being processed.
- `insideTag`: Tracks whether the function is processing inside a tag.
- `indentationLevel`: Keeps track of how much to indent the current line.

2.1.2.2 Process Each Character:

- **If the character is `<`:**
  - Starts a new tag.
  - If `currentTag` contains text content, it writes it to `output` with proper indentation.
  - Appends `<` to `currentTag` and sets `insideTag` to `true`.
- **If the character is `>`:**
  - Ends the current tag.
  - Checks whether the tag is:
    - **Closing tag** (e.g., `</tag>`): Reduces indentation and writes the tag.
    - **Self-closing tag** (e.g., `<tag/>`): Writes the tag with current indentation.
    - **Opening tag** (e.g., `<tag>`): Increases indentation after writing the tag and adds it to the stack.
  - Clears `currentTag`.
- **If inside a tag:**
  - Appends the character to `currentTag` until the tag is complete.
- **If outside tags:**
  - Handles text content between tags, ensuring excessive whitespace is trimmed.

2.1.2.3 Finalize Remaining Content:

- If there's leftover content in `currentTag` after the loop, it writes it to `output`.

2.1.2.4 Return the Result:

Returns the fully formatted XML string.
Time Complexity : $O(n)$

2.1.3 XML Conversion to JSON

This is an explanation of the file `xml2json.cpp`

2.1.3.1 Explanation of the Code

The code converts an XML string into JSON format using the `XmlToJsonConverter` class and provides functionality to save the resulting JSON to a file. Below is a detailed breakdown:

2.1.3.2 1. *Struct JsonNode*

- Represents a node in the JSON structure.
- **Attributes:**
  - `tag`: Stores the tag name of the XML element.
  - `value`: Stores the value inside an XML tag (text between `<tag>` and `</tag>`).
  - `children`: A vector of child `JsonNode`s to represent nested structures.
- **Method:**
  - `toJson(int indent = 0)`:
    - Converts the `JsonNode` and its children into a JSON-formatted string.
    - Handles indentation to format the output neatly.
    - Checks if the node has children:
      - If yes, formats it as a JSON object ({}).
      - Otherwise, outputs the value as a JSON string.

2.1.3.3 2. *Class XmlToJsonConverter*

- Main class to handle XML-to-JSON conversion.
- **Methods:**

1. `convertToJson(const string &xml)`:
   - Converts an XML string to JSON format.
   - **Variables:**
     * `stack<JsonNode> nodes`: Tracks the current hierarchy of nodes while parsing XML.
     * `root`: The root `JsonNode` representing the entire XML structure.
     * `tag` and `value`: Temporary variables to store the tag name and the value inside an XML element, respectively.
     * `insideTag` and `closingTag`: Flags to track if the parser is inside an XML tag or processing a closing tag (`</tag>`).
   - **Logic:**
     * Iterates through the XML string.
     * When encountering `<`:
       · Processes any value found before the tag (if applicable).
       · Marks the start of a new tag.
     * When encountering `>`:
       · If it's a closing tag (`</tag>`), pops the completed node from the stack and attaches it to its parent.
       · If it's an opening tag (`<tag>`), pushes a new node onto the stack.
     * Any content outside `<` and `>` is treated as value.
   - Returns the JSON representation of the root node.
2. `saveToFile(const string &json, const string &filename)`:
   - Writes the JSON string to a file.
   - If the file cannot be opened, prints an error message.
3. `trim(string &str)`:
   - Removes leading and trailing whitespace from a string.
   - Uses two pointers to find the non-whitespace start and end of the string.

2.1.3.4 3. *Main Flow*

The code takes an XML string, converts it to JSON format using `convertToJson()`, and saves it to a file if needed.

2.1.3.5 Time and Space Complexity

- **Time Complexity:**
  - `convertToJson()`:
    - Parsing the XML string is linear with respect to its size, i.e., $O(n)$, where n is the length of the XML string.
    - Constructing the JSON representation involves traversing all nodes:
      - Each node's `toJson()` method is called once. Since there are m nodes, this takes $O(m)$.
    - Overall: $O(n + m)$.
    - If the XML string size correlates with the number of nodes, it simplifies to $O(n)$.
  - `saveToFile()`:
    - Writing the JSON to a file is $O(k)$, where k is the size of the JSON string.
    - Combined with `convertToJson()`: $O(n + k)$.
- **Space Complexity:**
  - `convertToJson()`:
    - The stack used for parsing requires $O(h)$ space, where h is the height of the XML tag tree.
    - The root object stores the entire JSON structure in memory, which takes $O(m)$.
    - Total: $O(m + h)$.
  - `saveToFile()`:
    - Storing the JSON string requires $O(k)$.
    - Combined with `convertToJson()`: $O(m + k + h)$.

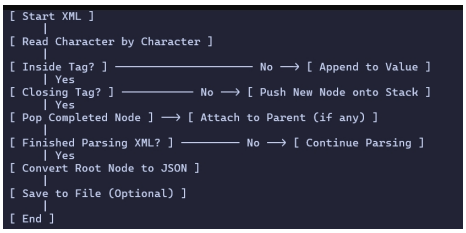

Figure 2.1: XML to JSON conversion process

2.1.4 Minifying the XML file

This is an explanation the file `Minifying.cpp` and header `Minifying.h`

2.1.4.1

The function `MinifyingFunction` processes an input string (like an HTML file) and reformats it by removing unnecessary whitespace while keeping tags and text content structured. Here's a step-by-step explanation:

2.1.4.2 Initialize Variables:

- `output` is an empty string where the processed result will be stored.
- `insideTag` is a boolean flag that tracks whether the current character is inside a tag (e.g., `<tag>`).

2.1.4.3 Loop Through Each Character:

The function iterates through every character in the input string:

- **If the character is `<`:**
  - Indicates the start of a tag.
  - Adds a newline before the tag if the `output` is not empty and doesn't already end with a newline.
  - Appends `<` to the `output` and sets `insideTag` to `true`.
- **If the character is `>`:**
  - Indicates the end of a tag.
  - Appends `>` to the `output`, sets `insideTag` to `false`, and adds a newline.
- **If inside a tag:**
  - Directly appends the character to the `output`.
- **If outside tags (text content):**
  - Removes excessive whitespace by ensuring only meaningful characters are added.
  - Adds text content to the `output` but avoids adding multiple consecutive spaces or newlines unnecessarily.

2.1.4.4 Return the Result:

After processing all characters, the function returns the reformatted string.

Time Complexity : $O(n)$

2.1.5 Compression/Decompression

This is an explanation of the file `compression.cpp` and header `compression.h` Compressing/Decompressing the XML file (or any kind of file) using Huffman Coding

2.1.5.1 Overview

Now we will understand the algorithm with a simple example and draw parallels from its steps to our code

Assume We have a simple file that we want to compress and this is its content:

"AABCBAD"

2.1.5.2 Reading the file contents

Time Complexity : Reading the file $O(n)$ where n is the number of characters in the file.
Space Complexity : Storing the file content in a buffer $O(n)$

2.1.5.3 Counting the Frequencies of each character

We will count how many times each character was repeated in the file

This is through using `vector<CharFrequency> frequencies` which is a vector of the struct `CharFrequency` which stores a character and its corresponding repetition frequency.

Character	Repetitions
A	3
B	2
C	1
D	1

Time Complexity : $O(n.f)$ where is the number of characters in the file and f is the number of unique characters
Space Complexity : $O(f)$ for storing the character-frequency pairs

2.1.5.4 Building the Huffman Tree

The Huffman tree consists of "`struct HuffmanNode`"s, we will learn its purpose and how to build it
We want to put the character-frequency pairs such that the least frequent is at the front, we will use a priority queue for that.

Step (1)

Figure 2.2: Pushing all character-frequency pairs into a priority queue

We take the least 2 frequencies in the queue through 2 `top()` operations and remove them from the queue using 2 `pop()` operations.
We then create a new `HuffmanNode` where its frequency is the sum of the two frequencies we just popped. Its left pointer points to the element of the larger frequency of the two nodes we popped and the right pointer points to the element of the lesser frequency.
Step (2)

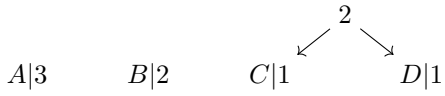

Figure 2.3: Creating a new Huffman Node

We then push the newly created Huffman Node into the priority queue.

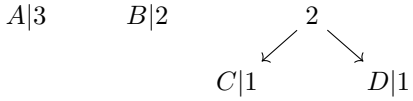

Figure 2.4: Pushing the new Huffman Node into the priority queue

Repeating Steps (1) and (2):

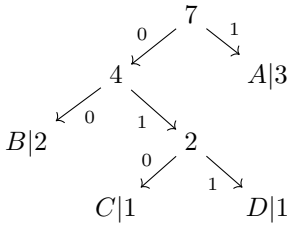

Figure 2.5: Now our priority queue contains only one element (7) which is our condition to stop. By convention left branch means a 0 and right means a 1

Time Complexity:

- Inserting f nodes into a priority queue : $O(f \log(f))$
- Merging $f - 1$ nodes during the tree-building process: $O(f \log(f))$
- Total : $O(f \log(f))$

Space Complexity: Each node occupies constant space, and there are $2f - 1$ nodes in the tree : $O(f)$

2.1.5.5 Generate the Huffman Codes

By convention each left branch traversal means a (0) and each right branch traversal means (1). Drawing the Huffman tree again:

We will then put the codes to the corresponding character.

Character	Repetitions	Code
A	3	1
B	2	00
C	1	010
D	1	011

Table 2.1: Table of Codes

Time Complexity : Traversing the Huffman Tree : $O(f)$ where f is the number of unique characters in the file.
Space Complexity : Storing f character-code pairs. Assuming the average code length is c: $O(fc)$

2.1.5.6 Encoding the Data

This is done using the `encode` function
Returning to our original file:

"AABCBAD"

Using the codes generated in Table 2.1

A	A	B	C	B	A	D
1	1	00	010	00	1	011

Figure 2.6: Result of Encoding

Time Complexity : For each character in the input, mapping it to its Huffman code takes $O(c)$, where c is the average code length.
Total for encoding : $O(n.c)$
Space Complexity : Compressed Data $O(m)$ where m represents the total number of bits required to store the encoded data after compression.

2.1.5.7 Writing the encoded data

We will begin by storing a header containing the number of unique characters and each unique character with their corresponding frequency. This is in order to decompress the compressed file without knowing the character-code pairs beforehand.
Time Complexity:

- Writing the header: $O(f)$
- Writing the compressed bits: $O(n)$
- Total : $O(f + n)$

Space Complexity:$O(m + f)$

2.1.5.8 Decompression

1. Reading the compressed file's header, we deduce the number of unique characters and each character's repetition count
2. Using this information we can build the Huffman tree to deduce the codes corresponding to each character

We then have the character-code pairs we need to decode the compressed file.
Time Complexity :

- Reading compressed file's header and data $O(m + f)$
- Rebuilding Huffman tree:
  - Inserting f nodes into a priority queue : $O(f \log(f))$
  - Merging $f - 1$ nodes during the tree-building process: $O(f \log(f))$
  - Total : $O(f \log(f))$
- Traversing Huffman tree:$O(m.h)$ where h is the height of the tree.
- Writing the Decompressed Data: $O(n)$

Space Complexity:

- Buffer for compressed file : $O(m + f)$
- Huffman tree :$O(2f - 1) = O(f)$
- Buffer for decompressed file : $O(n)$

2.1.5.9 Total Complexity

Time Complexity :

$$O(nf + f\log(f) + nc + mh)$$

Space Complexity:

$$O(n + fc + m)$$

2.2 Level (2)

Level (2) of the project is concerned with the graph operations mentioned in Section 1.1
It's in the file `Graph.cpp`

2.2.1 parseXML function

The `parseXML` function reads an XML file with user data, extracts key information (user ID, name, posts, followers), and stores it in a graph structure.

2.2.1.1 Parsing Tags

- **User Start Tag:** Detects the start of a user entry with `<user>`, resets the posts and followers lists, and sets `readingID` to `true`.
- **ID Parsing:** Extracts the user ID between `<id>` and `</id>` tags.
- **Name Parsing:** Extracts the user's name between `<name>` and `</name>` tags.
- **Post Parsing:** Sets `readingPosts` to `true` to extract posts between `<post>` and `</post>` tags.
- **Follower Parsing:** Sets `readingFollowers` to `true` to extract follower IDs between `<id>` and `</id>` tags.

2.2.1.2 Storing Data

After parsing, a `User` object is created and added to the graph using the `AddVertex` method.

2.2.1.3 Closing the File

The file is closed using `file.close()` once all data is read.

2.2.1.4 Adding Edges Between Users

The `addEdgesBetweenUsers` function establishes follower relationships by creating edges between users. If a follower is not found, a new "Unknown User" is created.

2.2.1.5 Time and Space Complexity:

Time Complexity

- **File Reading:** The function reads each line of the file once, so the time complexity for reading is $O(L)$, where L is the number of lines in the file.
- **Processing Lines:** String operations like `find()` and `substr()` take constant time for small XML tags, resulting in $O(1)$ per line.
- **Overall Time Complexity:** $O(L)$, where L is the number of lines in the file.

Space Complexity

- **User Data Storage:** For each user, the ID, name, posts, and followers are stored. The space complexity depends on the number of users (U) and their posts (P) and followers (F).
- **Overall Space Complexity:** $O(U \cdot (P + F))$, where U is the number of users, P is the average number of posts, and F is the average number of followers.

2.2.1.6 Flowchart

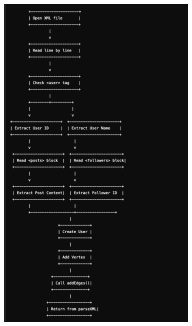

Figure 2.7: Flowchart for `parseXML` function

2.2.2 addEdgesBetweenUsers function

This function adds directed edges between users based on their follower relationships.

- **Loop Through All Users:** Iterates through all users in the graph (`vertices[]`) and checks their followers list (`Followers_id`).
- **Creating Edges:** For each follower ID in the user's followers list:
  - If the follower exists, an edge is created from the follower to the current user using `AddEdge`.

– If the follower does not exist, a new "Unknown User" is created and added to the graph, followed by creating an edge from the "Unknown User" to the current user.

2.2.2.1 Time and Space Complexity

Time Complexity

- The function loops through all users, and for each user, it iterates through their list of followers. Let U be the number of users and F be the average number of followers per user.
- For each follower, the function either finds the follower's index or creates a new user. Finding the index (`indexOf`) takes $O(U)$ time.
- **Overall Time Complexity:** The outer loop runs U times, and for each user, the inner loop runs F times, resulting in a time complexity of $O(U^2 \cdot F)$.

Space Complexity

- **Storing Edges:** The edges are stored in a 2D array (`edges[][]`), with a size of $U \times U$, so the space complexity for storing edges is $O(U^2)$.
- **Overall Space Complexity:** The space complexity is $O(U^2)$, where U is the number of users.

2.2.2.2 Flowchart

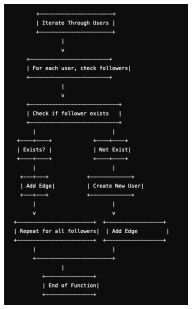

Figure 2.8: Flowchart for `addEdgesBetweenUsers` function

2.3 GUI/CLI Implementation

2.3.0.1 Overview

Having explained the project features and their workings, this section demonstrates how they are integrated into both a GUI and a CLI.

2.3.1 GUI

This part focuses on `xml_editor_GUI.py` and `Graph_GUI.py`. The graphical user interface (GUI) for the XML Editor was developed using Python's Tkinter library, enhanced with `ttkbootstrap` for a modern look. It allows users to interact with XML files, performing tasks like verifying, formatting, and converting XML files to JSON, among other features.

2.3.1.1 Displaying File Content

Handling large XML files without causing lag or unresponsiveness was challenging.

- **Solution:** A `Text` widget within a `Notebook` tab was used to display file content, separating it from operation logs for better data management.

2.3.1.2 Integrating Backend Commands

Connecting GUI buttons to backend operations, such as running a C++ executable, required careful management of input and output files.

- **Solution:** Python's `subprocess` module was employed to invoke the backend executable. File dialog prompts ensured proper file selection for input and output handling.

2.3.1.3 Error Handling and Feedback

The GUI initially lacked clear feedback for backend operation failures.

- **Solution:** Errors were displayed using `messagebox.showerror`, and operation logs were shown in a dedicated tab to improve user experience.

2.3.1.4 GUI Layout and Responsiveness

Ensuring the interface was clean, responsive, and organized despite numerous features was a design challenge.

- **Solution:** Buttons were grouped into labeled frames (e.g., "XML Operations" and "Graph Operations") and organized using a grid layout for a structured appearance.

The resulting GUI provides an intuitive way to interact with XML files and it's both versatile and user-friendly.

This project emphasized the complexities of integrating a graphical interface with backend operations.

2.3.2 CLI

2.3.2.1 Overview

The `xml_editor.cpp` file serves as the CLI core of the project, implementing all features from Section 1.1 in a command-line interface. It is as powerful as its GUI counterpart.

2.3.2.2 Command-Line Application Overview

The application processes commands via a structured input format: a command, an input file, and optional output files or flags. It validates arguments to ensure correctness before execution.

2.3.2.3 Argument Parsing

Arguments are parsed sequentially starting at index 2, recognizing flags and values:

- `-i`: Input file (mandatory).
- `-o`: Output file (mandatory for commands with outputs).
- `-f`: Fixes errors (e.g., in `verify`). No value required.
- Command-specific flags: Options like `-id` or `-ids` are dynamically parsed.

The program ensures proper flag-value pairing, e.g., `-i` must be followed by a valid file name.

2.3.2.4 Validation

Validation occurs at multiple levels:

1. **Command Check**: Verifies the first argument against valid commands.
2. **File Validation**: Ensures non-empty, valid paths for `-i` and `-o`.
3. **Flag Consistency**: Confirms required flags are present and correctly used.

2.3.2.5 Error Handling in Parsing

The program gracefully handles missing or invalid arguments. For example, if `-i` is missing, it reports: `"Error: Input file not specified. Use -i <input_file>."`

2.3.2.6 Dynamic Argument Handling

Commands with extra parameters like `-id` or `-ids` are processed dynamically, supporting flexible inputs such as lists or search terms.

2.3.2.7 Usage Instruction

If arguments are incorrect or incomplete, the program displays: `Usage: xml_editor <command> -i <input_file> [-o <output_file>] [options]`. This provides a clear reference for users.

References

[1] *DSA Huffman Coding*. https://www.w3schools.com/dsa/dsa_ref_huffman_coding.php.

[2] *Huffman Coding | Greedy Algo-3*. GeeksforGeeks. Section: Greedy. https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/.

[3] *Huffman coding - Wikipedia*. https://en.wikipedia.org/wiki/Huffman_coding.

[4] *Tutorial - ttkbootstrap*. https://ttkbootstrap.readthedocs.io/en/latest/gettingstarted/tutorial/.

[5] *Build Modern GUI Apps With TTKBootstrap, Tkinter, and Python | TutsNode*. July 7, 2023. https://tutsnode.org/build-modern-gui-apps-with-ttkbootstrap-tkinter-and-python/.

[6] *Run process with realtime output to a Tkinter GUI*. https://www.tutorialspoint.com/run-process-with-realtime-output-to-a-tkinter-gui.

[7] *Continuous Output Handling in Python Subprocesses – devgem.io - devgem.io*. https://www.devgem.io/posts/continuous-output-handling-in-python-subprocesses.

[8] Nick Lalic. *Terrabits/tkinter-multiprocessing-example*. original-date: 2019-01-23T20:32:02Z. Dec. 20, 2024. https://github.com/Terrabits/tkinter-multiprocessing-example.

[9] *Graphical User Interfaces with Tk*. Python documentation. https://docs.python.org/3/library/tk.html.

[10] Christian Göhring. *3 Ways To Parse Command Line Arguments in C++: Quick, Do-It-Yourself, Or Comprehensive*. Medium. Nov. 6, 2021. https://medium.com/@mostsignificant/3-ways-to-parse-command-line-arguments-in-c-quick-do-it-yourself-or-comprehensive-36913284460f.

[11] *Command Line Arguments in C++*. GeeksforGeeks. Section: C++. https://www.geeksforgeeks.org/command-line-arguments-in-cpp/.

[12] *JSON vs XML*. https://www.w3schools.com/js/js_json_xml.asp.