AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING

DSA Project

SocialX XML Editor

Data Structures and Algorithms

Fall 2025

Team Members

Name	ID
Ibrahim Mahmoud	2200182
Youssef Yacoub	2200649
Mohamed Khaled Ahmed	2201057
Ahmed Nagger Hassan	2200306
Mosa Abdulaziz	2200257
Mostafa Al Hassan Salah	2200747
Bassam Hussam	2200084
Youssef Medhat Mahmoud	2200626
Omar Mohamed Mahmoud Eid	2200284
Omar Mohsen Mohamed Amin	2200418

Faculty of Engineering
Ain Shams University

Contents

System Overview

1.1 Entry Point Architecture

The SocialX XML Editor follows a layered architecture pattern with clear separation between the entry point, user interfaces, and business logic. The system provides dual interface support through both a Graphical User Interface (GUI) and a Command Line Interface (CLI).

1.1.1 Main Entry Point (main.py)

The `main.py` file serves as the central routing mechanism for the entire application. It implements the following responsibilities:

- **Banner Display**: Renders ASCII art branding on startup
- **Menu System**: Provides a numbered menu for mode selection
- **Mode Routing**: Delegates to GUI or CLI based on user choice
- **Lifecycle Management**: Handles graceful exit and keyboard interrupts

Listing 1.1: Main Application Flow

```
def app():
    print_banner()
    print_help()
    while True:
        choice = input("Enter your choice (1/2/3/4): ")
        if choice == '1':
            launch_gui()     # Delegates to gui.py
        elif choice == '2':
            launch_cli()     # Delegates to cli.py
        elif choice == '3':
            print_help()
        elif choice == '4':
            sys.exit(0)
```

Figure 1.1: Entry Point

1.1.2 GUI Module (gui.py)

The GUI is managed by the `AppManager` class which implements a window navigation controller pattern:

- **QApplication Singleton**: Reuses existing Qt application instance to prevent crashes
- **Window Management**: Controls transitions between Landing, Browse, and Manual windows
- **Signal-Slot Architecture**: Uses Qt signals for decoupled navigation

Listing 1.2: AppManager Architecture

```
class AppManager:
    def __init__(self):
        self.app = QApplication.instance() or QApplication(sys.
            argv)
        self.landing_window = LandingWindow()
        self.manual_window = None
        self.browse_window = None
        # Connect navigation signals
        self.landing_window.browse_clicked.connect(self.
            show_browse_mode)
        self.landing_window.manual_clicked.connect(self.
            show_manual_mode)
```

Figure 1.2: GUI Module (Landing Window)

1.1.3 CLI Module (cli.py)

The CLI implements both single-command execution and an interactive REPL (Read-Eval-Print Loop):

- **Argument Parser Factory**: Creates configured `argparse` parsers
- **Command Executor**: Pattern-matched command dispatch using Python's `match` statement
- **REPL Loop**: Interactive mode with bash-style prompts and command history

Listing 1.3: CLI Command Dispatch

```
def execute_command(args, editor, graph):
    match args.command:
        case 'verify':
            # XML validation logic
        case 'format':
            # XML formatting logic
        case 'compress':
            # Compression logic
        # ... additional commands
```

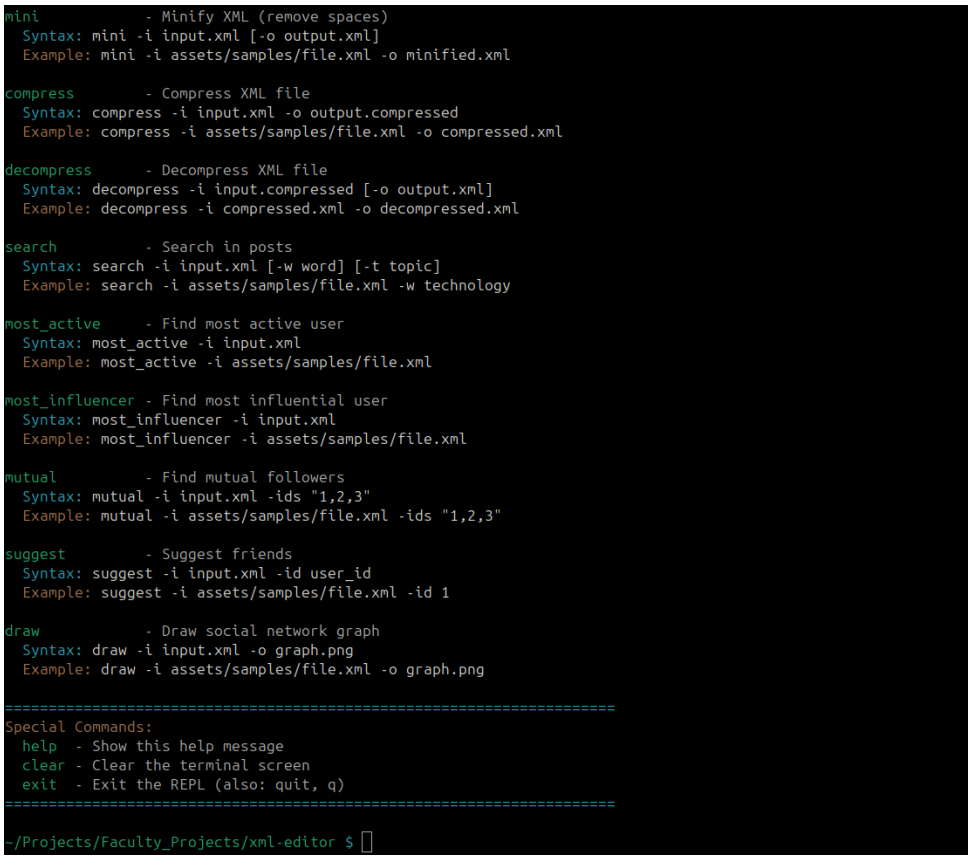

Figure 1.3: CLI Module

1.2 Architectural Diagram

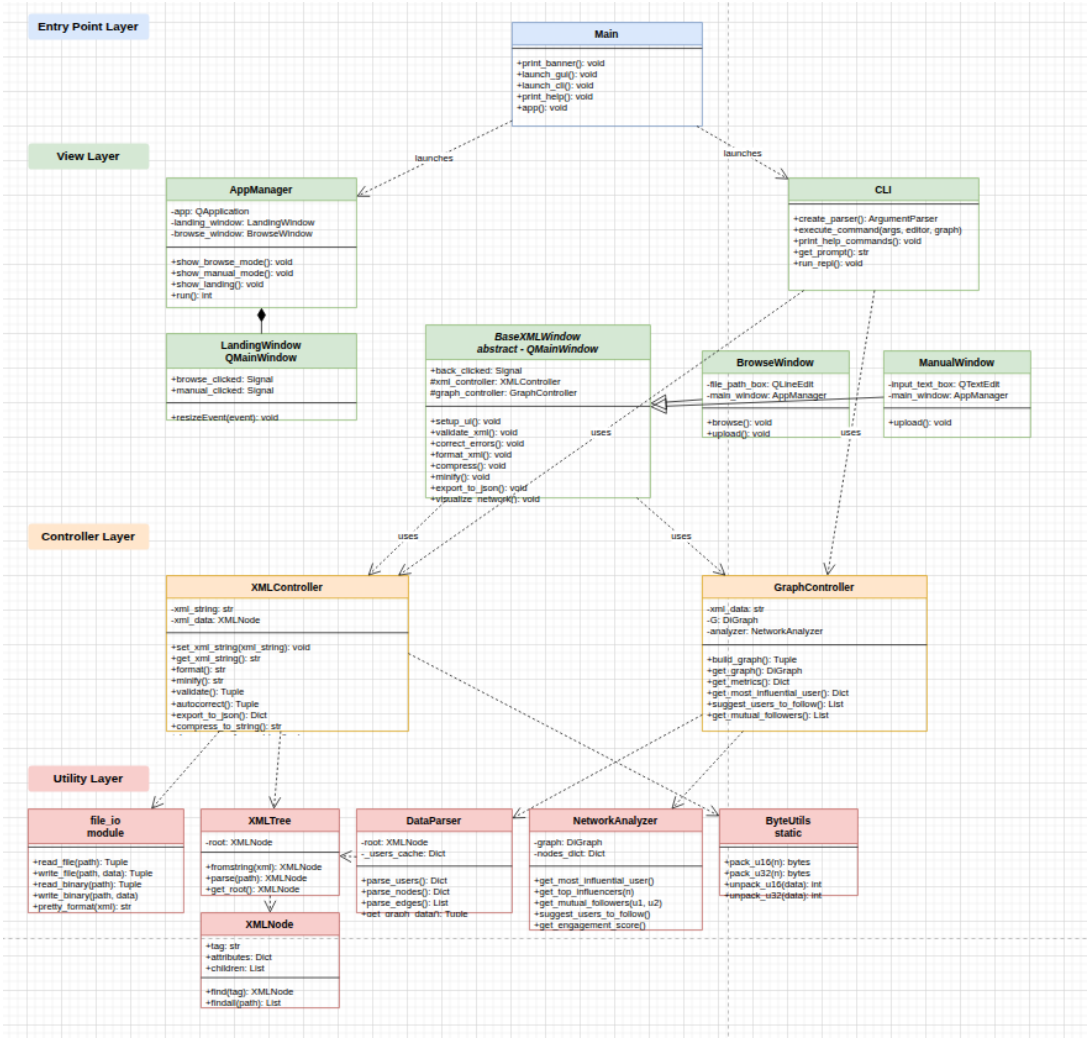

Figure 1.4: System Architecture Overview

Module Logic & Complexity Analysis

This chapter provides detailed analysis of each module in the `src/` folder, explaining how business logic is decoupled from the UI, and providing Big O complexity analysis for core functions.

2.1 Controllers Layer

2.1.1 XMLController (xml_controller.py)

The `XMLController` is the primary interface for all XML processing operations. It maintains internal state (`xml_string`) and provides methods that can be called from either GUI or CLI without modification.

Tokenization

Function: `_get_tokens()`

Algorithm: Linear scan tokenizer that splits XML into tags and text content.

Listing 2.1: Tokenization Algorithm

```
def _get_tokens(self) -> List[str]:
    tokens = []
    i = 0
    while i < len(self.xml_string):
        if self.xml_string[i] == '<':
            j = self.xml_string.find('>', i)
            tokens.append(self.xml_string[i:j+1])
            i = j + 1
        else:
            # Extract text content
            ...
    return tokens
```

Metric	Complexity	Notes
Time Complexity	$O(n)$	Single pass through input
Space Complexity	$O(n)$	Stores all tokens

Validation

Function: `validate()`

Algorithm: Stack-based XML structure validation using regex for tag extraction.

Data Structure: Stack to track opening tags and their line indices.

Listing 2.2: validate() core implementation

```python
def validate(self) -> Tuple[str, Dict[str, int]]:
    stack = []
    orphan_count = mismatch_count = missing_count = 0
    lines = self.xml_string.split('\n')
    annotated_lines = lines[:]

    for line_idx, line in enumerate(lines):
        tags = re.finditer(r'<(/?)(\w+)[^>]*>', line)
        for match in tags:
            is_closing = match.group(1) == '/'
            tag_name = match.group(2)
            if not is_closing:
                stack.append({'tag': tag_name, 'line_idx':
                    line_idx})
            else:
                if not stack:
                    orphan_count += 1
                    annotated_lines[line_idx] += f" <--- ORPHAN
                        TAG: </{tag_name}>"
                else:
                    top = stack[-1]
                    if top['tag'] == tag_name:
                        stack.pop()
                    else:
                        mismatch_count += 1
                        annotated_lines[line_idx] += f" <--- 
                            MISMATCH: Expected </{top['tag']}>, 
                            found </{tag_name}>."

    while stack:
        missing_count += 1
        leftover = stack.pop()
        annotated_lines[leftover['line_idx']] += f" <--- MISSING
            CLOSING TAG: <{leftover['tag']}>"

    error_counts = {
        'orphan_tags': orphan_count,
        'mismatches': mismatch_count,
        'missing_closing_tags': missing_count,
        'total': orphan_count + mismatch_count + missing_count
    }
    annotated_string = "\n".join(annotated_lines)
    return annotated_string, error_counts
```

Metric	Complexity	Notes
Time Complexity	$O(n \cdot m)$	n = lines, m = avg tags per line
Space Complexity	$O(d)$	d = maximum nesting depth

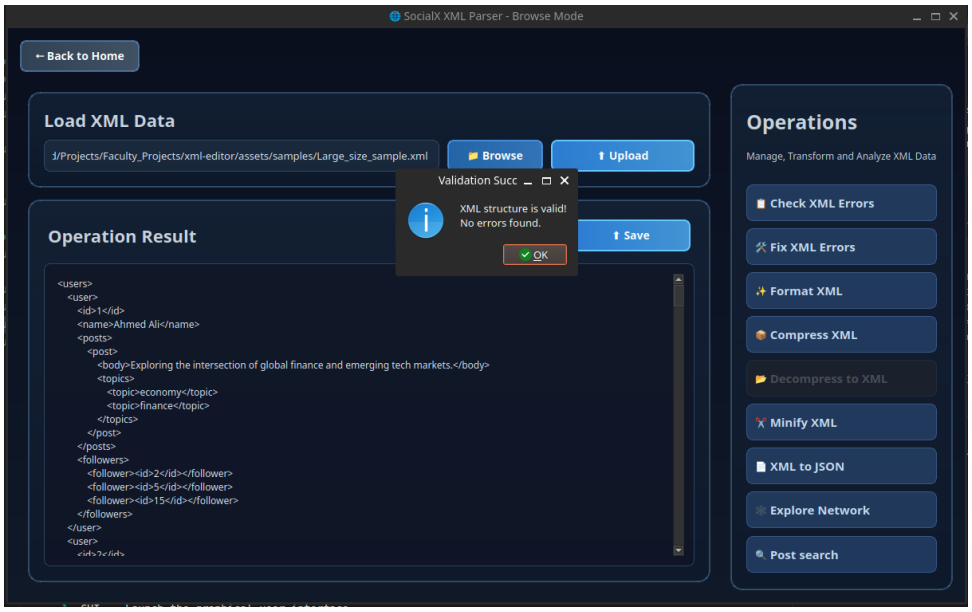

Figure 2.1: Validation a Large XML File

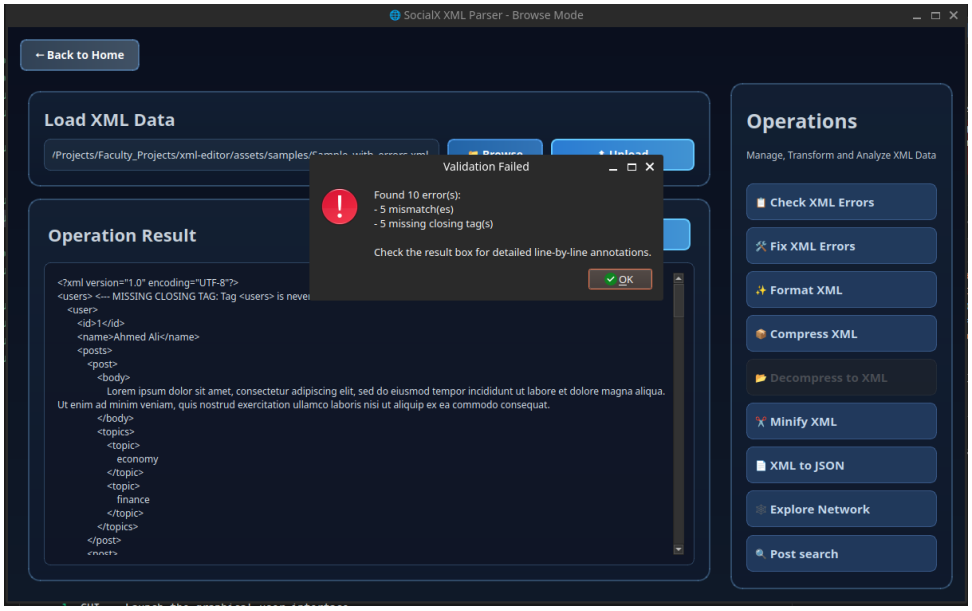

Figure 2.2: Validation an Error XML File

Auto-Correction

Function: `autocorrect()`

Algorithm: Stack-based tag balancing with lookahead for mismatched tags.

Listing 2.3: autocorrect() core implementation

```
def autocorrect(self) -> Tuple[str, Dict[str, int]]:
    stack = []
    missing_tags_added = stray_tags_removed = mismatches_fixed = 0
    tokens = re.split(r'(<[^>]+>)', self.xml_string)
    corrected_output = []
```

```
6
7       for token in tokens:
8           if not token:
9               continue
10          match = re.match(r'<(/?)(\w+)[^>]*>', token)
11          if match:
12              is_closing = match.group(1) == '/'
13              tag_name = match.group(2)
14              if not is_closing:
15                  stack.append(tag_name)
16                  corrected_output.append(token)
17              else:
18                  if stack and stack[-1] == tag_name:
19                      stack.pop()
20                      corrected_output.append(token)
21                  else:
22                      if tag_name in stack:
23                          while stack[-1] != tag_name:
24                              missing = stack.pop()
25                              corrected_output.append(f"</{missing}>
                                    ")
26                              mismatches_fixed += 1
27                          stack.pop()
28                          corrected_output.append(token)
29                      else:
30                          stray_tags_removed += 1
31          else:
32              corrected_output.append(token)
33
34      while stack:
35          missing = stack.pop()
36          corrected_output.append(f"</{missing}>")
37          missing_tags_added += 1
38
39      self.xml_string = "".join(corrected_output)
40      correction_counts = {
41          'missing_tags_added': missing_tags_added,
42          'stray_tags_removed': stray_tags_removed,
43          'mismatches_fixed': mismatches_fixed,
44          'total_corrections': missing_tags_added +
               stray_tags_removed + mismatches_fixed
45      }
46      return self.xml_string, correction_counts
```

Metric	Complexity	Notes
Time Complexity	$O(n \cdot d)$	d = nesting depth for stack search
Space Complexity	$O(n)$	Stores corrected output

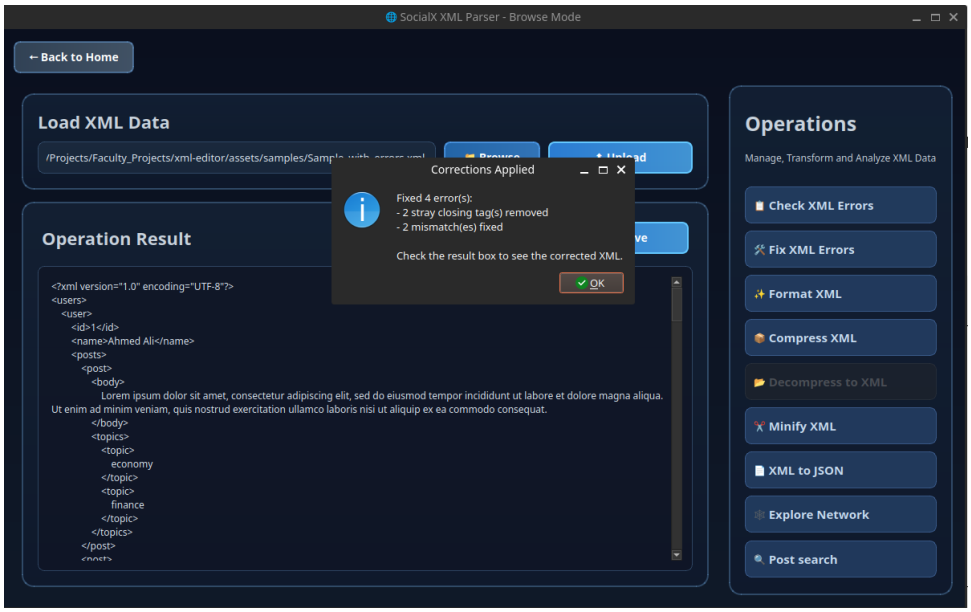

Figure 2.3: Auto-Correction a Large XML File with errors

Formatting

Function: `format()`

Algorithm: Tree-based indentation using token lookahead for inline text detection.

Listing 2.4: format() core implementation

```
def format(self) -> str:
    tokens = self._get_tokens()
    formatted = []
    level = 0
    indentation = "␣␣␣␣"
    k = 0
    MAX_WIDTH = 80

    while k < len(tokens):
        token = tokens[k]
        if token.startswith('</'):
            level = max(0, level - 1)
            formatted.append((indentation * level) + token)
        elif token.startswith('<') and not token.startswith('</'):
            if (k + 2 < len(tokens) and
                    not tokens[k + 1].startswith('<') and
                    tokens[k + 2].startswith('</')):
                text_content = tokens[k + 1]
                clean_text = "␣".join(text_content.split())
                if len(clean_text) > MAX_WIDTH:
                    formatted.append((indentation * level) +
                        tokens[k])
                    wrapper = textwrap.TextWrapper(width=MAX_WIDTH
                        , break_long_words=False)
                    for line in wrapper.wrap(clean_text):
```

```
                        formatted.append((indentation * (level +
                            1)) + line)
                    formatted.append((indentation * level) +
                        tokens[k + 2])
                else:
                    line = (indentation * level) + tokens[k] +
                        clean_text + tokens[k + 2]
                    formatted.append(line)
                k += 2
            else:
                formatted.append((indentation * level) + token)
                level += 1
        else:
            formatted.append((indentation * level) + token.strip()
                )
        k += 1
    return "\n".join(formatted)
```

Metric	Complexity	Notes
Time Complexity	$O(n)$	Single pass with lookahead
Space Complexity	$O(n)$	Stores formatted output

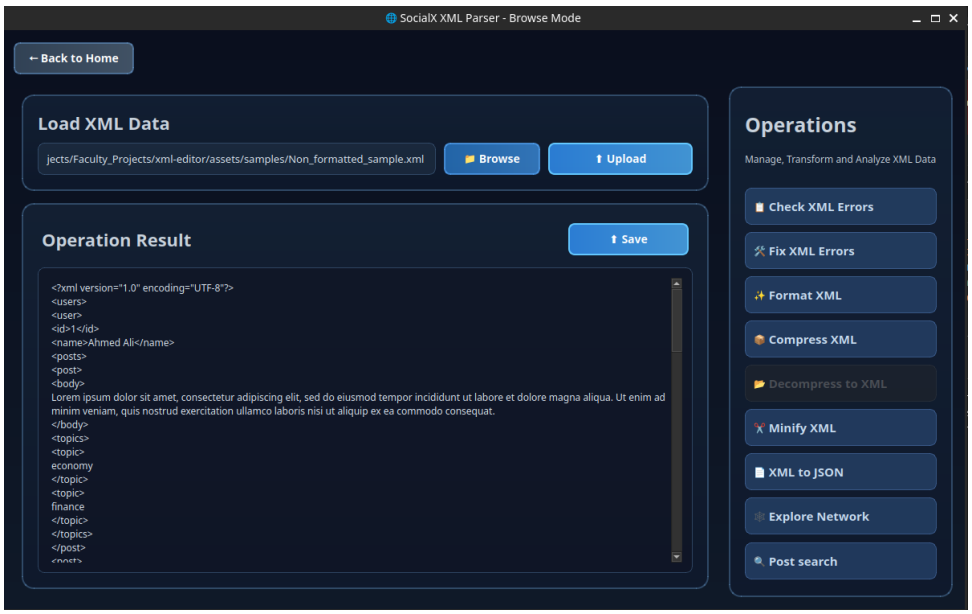

Figure 2.4: Not Formatted XML File

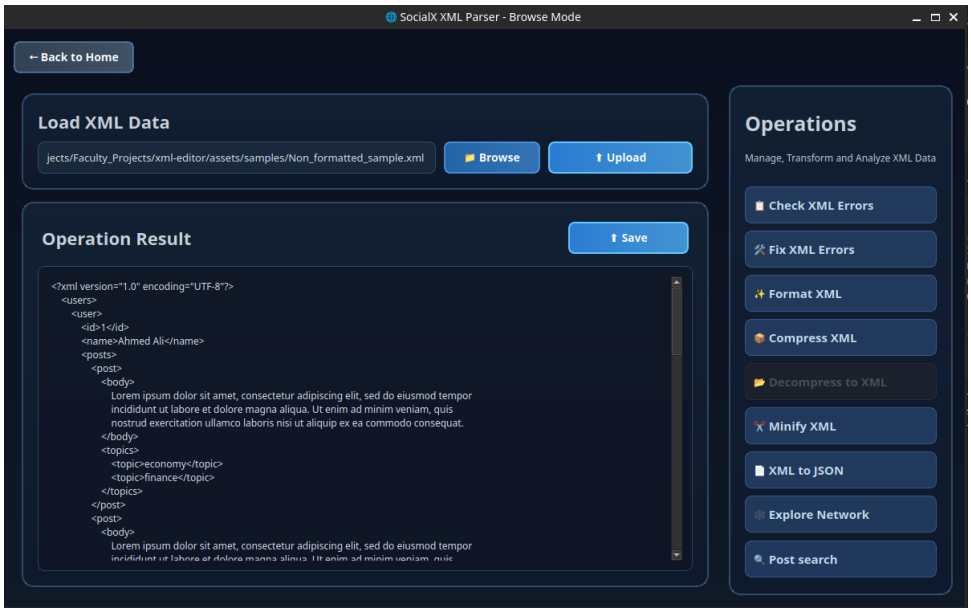

Figure 2.5: Formatted XML File

Minification

Function: `minify()`

Algorithm: Token concatenation without whitespace.

Listing 2.5: minify() core implementation

```
def minify(self) -> str:
    tokens = self._get_tokens()
    return "".join(tokens)
```

Metric	Complexity	Notes
Time Complexity	$O(n)$	Tokenize + join
Space Complexity	$O(n)$	Output string

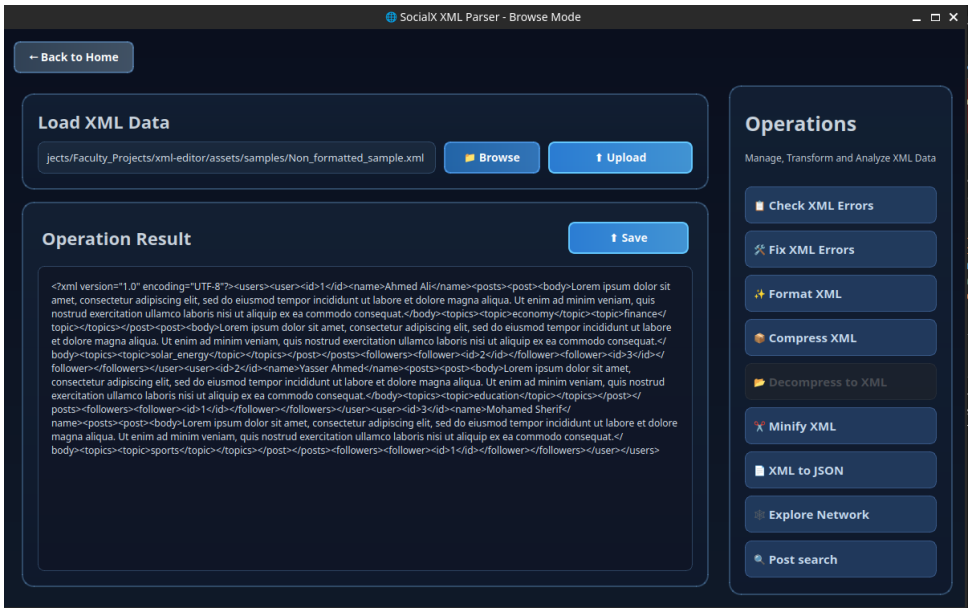

Figure 2.6: Minified XML File

Compression (BPE Algorithm)

Function: `compress_to_string()`

Algorithm: Byte Pair Encoding (BPE) - iteratively replaces the most frequent adjacent token pairs with a new token.

Listing 2.6: BPE Compression

```
def compress_to_string(self):
    tokens = [ord(c) for c in self.xml_string]
    merges = []
    next_token = 256

    for _ in range(100):  # Max 100 merge iterations
        # Count pair frequencies
        pair_counts = {}
        for i in range(len(tokens) - 1):
            key = (tokens[i] << 16) | tokens[i + 1]
            pair_counts[key] = pair_counts.get(key, 0) + 1

        # Find most frequent pair
        most_key = max(pair_counts, key=pair_counts.get)

        # Merge all occurrences
        merges.append((most_key, next_token))
        tokens = merge_tokens(tokens, most_key, next_token)
        next_token += 1
```

Metric	Complexity	Notes
Time Complexity	$O(k \cdot n)$	k = iterations (max 100), n = token count
Space Complexity	$O(n + k)$	Tokens + merge table

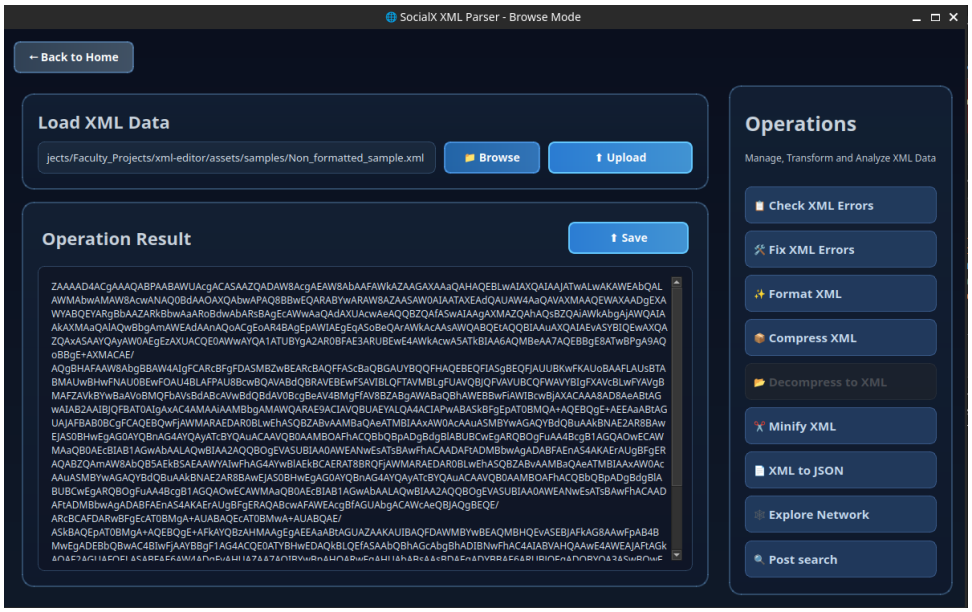

Figure 2.7: BPE Compression

Decompression

Function: `decompress_from_string()`

Algorithm: Reverse BPE expansion in reverse merge order.

Listing 2.7: `decompress_from_string()` core implementation

```
def decompress_from_string(self,
                           output_path: Optional[str] = None,
                           input_path: Optional[str] = None,
                           compressed_string: Optional[str] = None
                           ) -> str:
    if input_path is not None:
        data = bytearray(base64.b64decode(Path(input_path).
            read_text().strip()))
    elif compressed_string is not None:
        data = bytearray(base64.b64decode(compressed_string.strip
            ()))
    else:
        raise ValueError("You must provide either an input_path or
            a compressed_string.")

    offset = 0
    merge_count = ByteUtils.unpack_u32(data, offset); offset += 4
    merges = []
    for _ in range(merge_count):
        t1 = ByteUtils.unpack_u16(data, offset); offset += 2
        t2 = ByteUtils.unpack_u16(data, offset); offset += 2
        merged = ByteUtils.unpack_u16(data, offset); offset += 2
        merges.append((merged, t1, t2))

    token_count = ByteUtils.unpack_u32(data, offset); offset += 4
    tokens = []
```

```
    for _ in range(token_count):
        tokens.append(ByteUtils.unpack_u16(data, offset))
        offset += 2

    for merged_token, t1, t2 in reversed(merges):
        new_tokens = []
        for t in tokens:
            if t == merged_token:
                new_tokens.extend([t1, t2])
            else:
                new_tokens.append(t)
        tokens = new_tokens

    output = ''.join(chr(t) for t in tokens)
    if output_path is not None:
        file_io.write_file(output_path, data=output)
    return output
```

Metric	Complexity	Notes
Time Complexity	$O(k \cdot n)$	k = merges, n = final output size
Space Complexity	$O(n)$	Expanded tokens

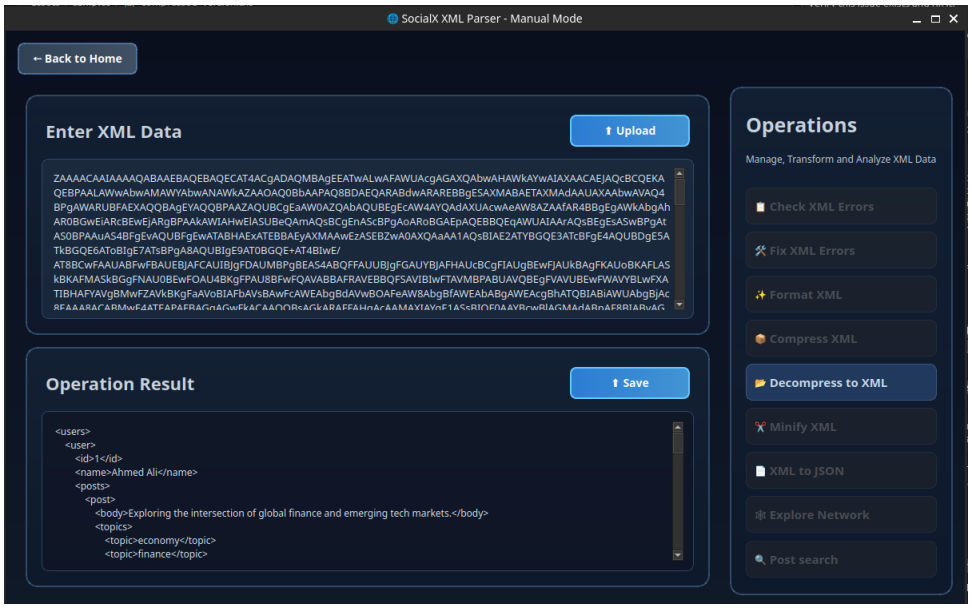

Figure 2.8: BPE Decompression

JSON Export

Function: `export_to_json()`

Algorithm: State machine parser with parent stack for context tracking.

Listing 2.8: `export_to_json()` core implementation

```
def export_to_json(self) -> dict[str, list[Any]]:
    tokens = self._get_tokens()
```

```
    json_data = {"users": []}
    user_dict = post_dict = relationship_dict = None
    current_container = None
    parent_stack = []
    i = 0
    while i < len(tokens):
        token = tokens[i]
        if token.startswith('<') and not token.startswith('</'):
            tag_name, attrs = self._get_tag_info(token)
            if tag_name == 'user':
                user_dict = {"id": attrs.get('id'), "name": None,
                    "posts": [], "followers": [], "followings": []}
            elif tag_name == 'post' and user_dict is not None:
                post_dict = {"content": None, "topics": []}
            elif tag_name in ('follower', 'following'):
                relationship_dict = {}
            parent_stack.append(tag_name)
            current_container = tag_name
        elif token.startswith('</'):
            tag_name, _ = self._get_tag_info(token)
            if tag_name == 'user' and user_dict is not None:
                json_data["users"].append(user_dict); user_dict =
                    None
            elif tag_name == 'post' and post_dict is not None and
                user_dict is not None:
                user_dict["posts"].append(post_dict); post_dict =
                    None
            elif tag_name == 'follower' and relationship_dict is
                not None and user_dict is not None:
                user_dict["followers"].append(relationship_dict);
                    relationship_dict = None
            elif tag_name == 'following' and relationship_dict is
                not None and user_dict is not None:
                user_dict["followings"].append(relationship_dict);
                    relationship_dict = None
            if parent_stack and parent_stack[-1] == tag_name:
                parent_stack.pop()
            current_container = None
        else:
            text_content = token.strip()
            if not text_content:
                i += 1; continue
            if current_container == 'name' and user_dict is not
                None and user_dict["name"] is None:
                user_dict["name"] = text_content
            elif current_container in ('body', 'content') and
                post_dict is not None and post_dict["content"] is
                None:
                post_dict["content"] = text_content
            elif current_container == 'topic' and post_dict is not
                None:
```

```
                post_dict["topics"].append(text_content)
            elif current_container == 'id':
                parent_tag = parent_stack[-2] if len(parent_stack)
                    >= 2 else None
                if parent_tag in ('follower', 'following') and
                   relationship_dict is not None:
                    relationship_dict["id"] = text_content
                elif parent_tag == 'user' and user_dict is not
                   None and user_dict["id"] is None:
                    user_dict["id"] = text_content
            current_container = None
        i += 1
    return json_data
```

Metric	Complexity	Notes
Time Complexity	$O(n)$	Single pass through tokens
Space Complexity	$O(u+p)$	u = users, p = posts

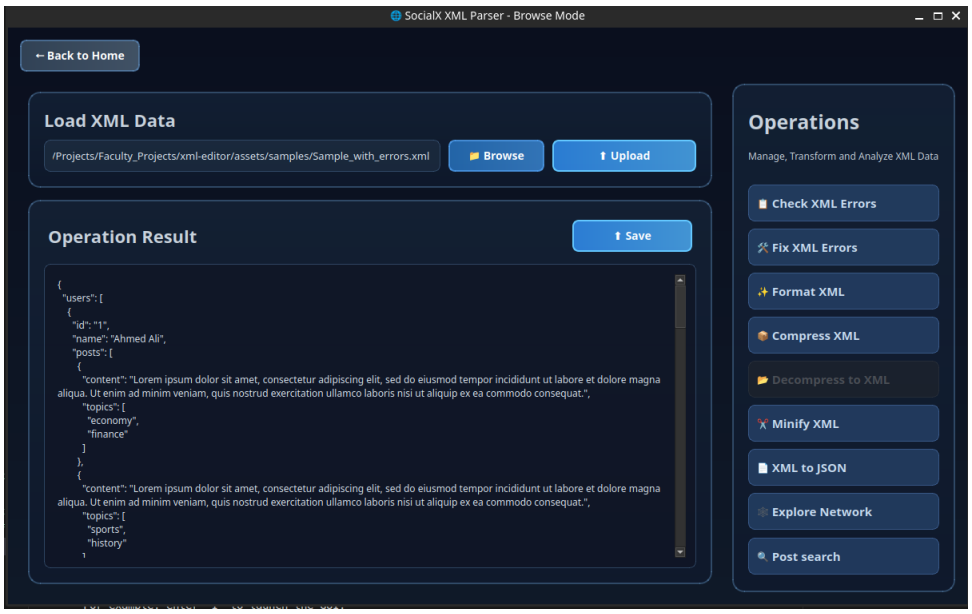

Figure 2.9: JSON Export

Post Search

Function: `search_in_posts()`

Algorithm: Linear search through XML tree nodes with substring matching.

Listing 2.9: `search_in_posts()` core implementation

```
def search_in_posts(self,
                    word: Optional[str] = None,
                    topic: Optional[str] = None
                    ) -> Optional[List[str]]:
    if (word is None and topic is None) or (word is not None and
       topic is not None):
```

```
        return None
    if hasattr(self, 'xml_string') and self.xml_data is None:
        self.xml_data = XMLTree.fromstring(self.xml_string)
    if not self.xml_data:
        return None

    result = []
    users = self.xml_data.findall('.//user')
    for user in users:
        name_node = user.find('name')
        user_name = name_node.text.strip() if (name_node and
            name_node.text) else "Unknown User"
        posts = user.findall('.//post')
        for post_elem in posts:
            found = False
            body_node = post_elem.find('body')
            body_text = body_node.text if (body_node and body_node
                .text) else ""
            if word is not None:
                if word.lower() in body_text.lower():
                    found = True
            elif topic is not None:
                topic_elements = post_elem.findall('.//topic')
                for topic_elem in topic_elements:
                    if topic_elem.text and topic.lower() in
                        topic_elem.text.lower():
                        found = True
                        break
            if found:
                clean_body = body_text.strip().replace('\n', ' ')
                result.append(f"in user: {user_name}'s. found
                    relevant post: {clean_body}\n\n")
    if len(result) == 0:
        result.append("found no relevant posts in any user's posts
            ")
    return result
```

Metric	Complexity	Notes
Time Complexity	$O(u \cdot p \cdot m)$	u = users, p = posts, m = text length
Space Complexity	$O(r)$	r = matching results

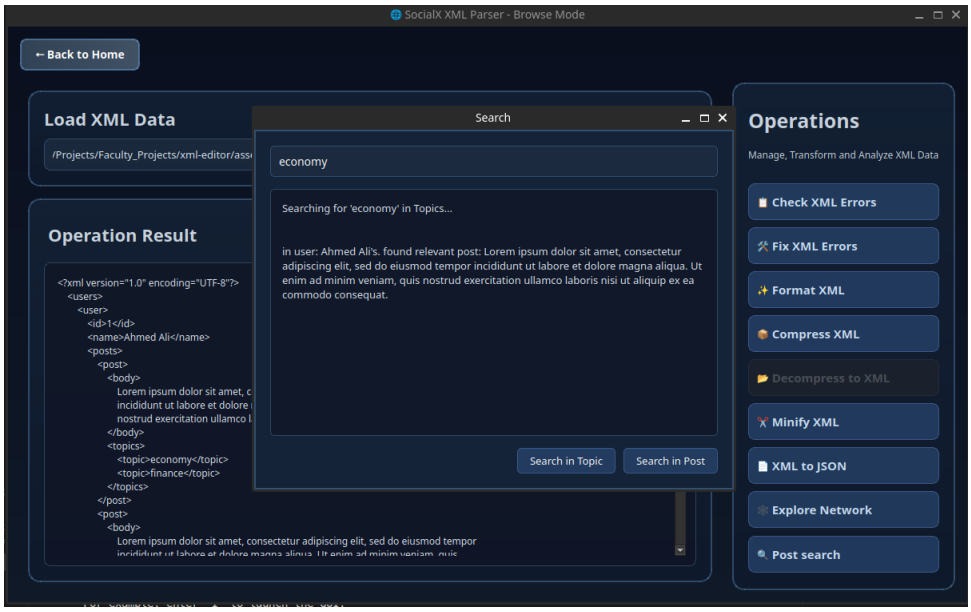

Figure 2.10: Post Search

2.1.2 GraphController (graph_controller.py)

The `GraphController` manages social network graph construction and analysis. It delegates to `NetworkAnalyzer` for advanced algorithms.

Graph Building

Function: `build_graph()`

Algorithm: Parse XML using `DataParser`, construct NetworkX DiGraph.

Listing 2.10: `build_graph()` core implementation

```
def build_graph(self) -> Tuple[bool, Dict[str, str], List[Tuple[
    str, str]], Optional[str]]:
    if self.xml_data is None:
        return False, {}, [], "No data loaded. Please upload and
            parse an XML file first."
    try:
        parser = DataParser(self.xml_data)
        nodes, edges = parser.get_graph_data()
        is_valid, errors = parser.validate_data()
        if not is_valid:
            print(f"Data validation warnings: {errors}")
        if len(nodes) == 0:
            return False, {}, [], "No users found in XML data."
        self.G = self._build_networkx_graph(nodes, edges)
        self.nodes_dict = nodes
        self.analyzer = NetworkAnalyzer(self.G, self.nodes_dict)
        self.metrics = self._calculate_metrics(nodes)
        return True, nodes, edges, None
    except Exception as e:
        return False, {}, [], f"Error building graph: {str(e)}"
```

Metric	Complexity	Notes
Time Complexity	$O(V + E)$	V = nodes, E = edges
Space Complexity	$O(V + E)$	Graph storage

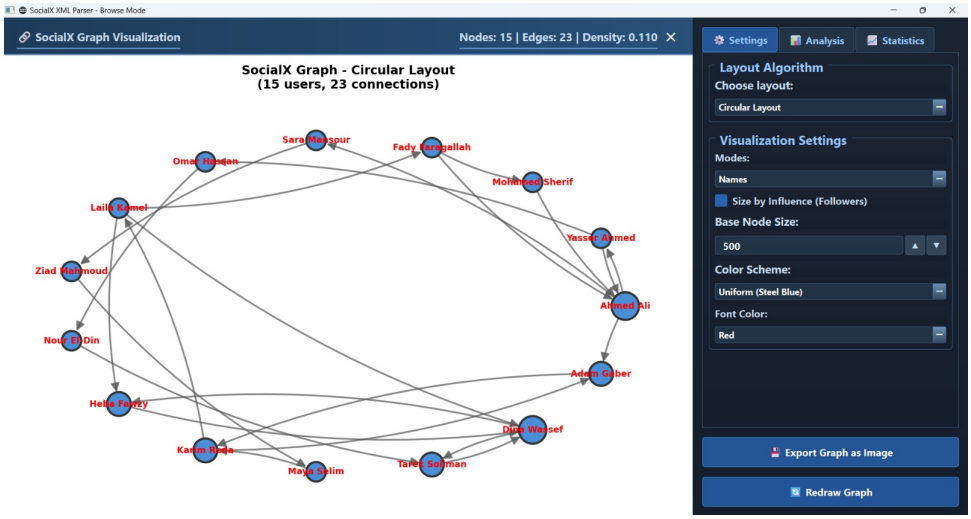

Figure 2.11: Graph Building

Metrics Calculation

Function: `_calculate_metrics()`

Algorithm: NetworkX degree calculations with numpy aggregation.

Listing 2.11: `_calculate_metrics()` core implementation

```
def _calculate_metrics(self, nodes: Dict[str, str]) -> Dict:
    if self.G is None:
        return {}
    metrics = {}
    metrics['num_nodes'] = self.G.number_of_nodes()
    metrics['num_edges'] = self.G.number_of_edges()
    metrics['density'] = nx.density(self.G)

    in_degrees = dict(self.G.in_degree())
    out_degrees = dict(self.G.out_degree())
    metrics['avg_in_degree'] = np.mean(list(in_degrees.values()))
       if in_degrees else 0
    metrics['avg_out_degree'] = np.mean(list(out_degrees.values())
       ) if out_degrees else 0

    if in_degrees:
        max_followers = max(in_degrees.values())
        metrics['most_influential'] = [
            {'id': uid, 'name': nodes.get(uid, 'Unknown'), '
               followers': in_degrees[uid]}
            for uid in in_degrees if in_degrees[uid] ==
               max_followers
        ]
```

```
    if out_degrees:
        max_following = max(out_degrees.values())
        metrics['most_active'] = [
            {'id': uid, 'name': nodes.get(uid, 'Unknown'), '
               following': out_degrees[uid]}
            for uid in out_degrees if out_degrees[uid] ==
               max_following
        ]
    metrics['in_degrees'] = in_degrees
    metrics['out_degrees'] = out_degrees
    return metrics
```

Metric	Complexity	Notes
Time Complexity	$O(V)$	Iterate all vertices
Space Complexity	$O(V)$	Degree dictionaries

2.2 Utilities Layer

2.2.1 XMLTree (xml_tree.py)

Custom XML parser implementing a tree data structure without using Python's built-in ElementTree.

XML Parsing

Function: `fromstring()`
Algorithm: Recursive descent parser with regex-based tag matching.
Data Structure: `XMLNode` tree with parent pointers and children lists.

Listing 2.12: XMLNode Structure

```
class XMLNode:
    def __init__(self, tag, attributes=None, text=None):
        self.tag = tag
        self.attributes = attributes or {}
        self.text = text
        self.children = []
        self.parent = None
```

Listing 2.13: fromstring() core implementation

```
@staticmethod
def fromstring(xml_string: str) -> XMLNode:
    parser = XMLTree()
    return parser._parse_string(xml_string)

def _parse_string(self, xml_string: str) -> XMLNode:
    xml_string = re.sub(r'<\?xml[^?]*\?>', '', xml_string).strip()
    xml_string = re.sub(r'<!--.*?-->', '', xml_string, flags=re.
       DOTALL)
    self.root = self._parse_element(xml_string)
```

```
10      return self.root
```

Metric	Complexity	Notes
Time Complexity	$O(n \cdot d)$	n = input size, d = depth
Space Complexity	$O(n)$	Tree nodes

Recursive Search

Function: `findall('.//tag')`
Algorithm: Depth-first search through tree.

Listing 2.14: findall() core implementation

```
1  def findall(self, path: str) -> List['XMLNode']:
2      if path.startswith('.//'):
3          tag = path[3:]
4          return self._find_recursive(tag)
5      else:
6          return [child for child in self.children if child.tag ==
               path]
7
8  def _find_recursive(self, tag: str) -> List['XMLNode']:
9      results = []
10     for child in self.children:
11         if child.tag == tag:
12             results.append(child)
13         results.extend(child._find_recursive(tag))
14     return results
```

Metric	Complexity	Notes
Time Complexity	$O(n)$	Visit all nodes
Space Complexity	$O(d)$	Recursion stack

2.2.2 NetworkAnalyzer (network_analyzer.py)

Advanced graph analysis algorithms for social network features.

Most Influential User

Function: `get_most_influential_user()`
Algorithm: Find maximum in-degree (follower count).

Listing 2.15: `get_most_influential_user()` core implementation

```
1  def get_most_influential_user(self) -> Optional[Dict]:
2      """
3      Get the user with the most followers (highest in-degree).
4
5      Returns:
6          Dict with user_id, name, and followers count, or None if
               graph is empty
```

```
    """
    if self.G.number_of_nodes() == 0:
        return None

    in_degrees = dict(self.G.in_degree())
    if not in_degrees:
        return None

    most_influential_id = max(in_degrees, key=in_degrees.get)

    return {
        'user_id': most_influential_id,
        'name': self.nodes_dict.get(most_influential_id,
            most_influential_id),
        'followers': in_degrees[most_influential_id]
    }
```

Metric	Complexity	Notes
Time Complexity	$O(V)$	Linear scan of degrees
Space Complexity	$O(V)$	Degree dictionary

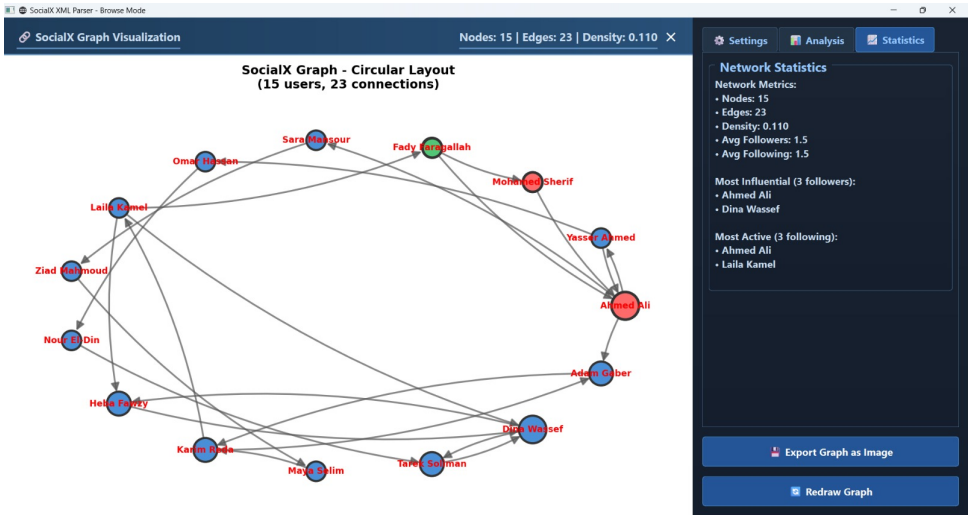

Figure 2.12: Most Influential User

Mutual Followers

Function: `get_mutual_followers_between_many()`
Algorithm: Set intersection of predecessor sets.

Listing 2.16: Mutual Followers Algorithm

```
def get_mutual_followers_between_many(self, user_ids):
    mutual = set(self.G.predecessors(user_ids[0]))
    for user_id in user_ids[1:]:
        user_followers = set(self.G.predecessors(user_id))
        mutual &= user_followers  # Set intersection
```

```
    return [{'user_id': f, 'name': self.nodes_dict[f]} for f in
        mutual]
```

Metric	Complexity	Notes
Time Complexity	$O(k \cdot \min(f_i))$	k = users, f_i = followers per user
Space Complexity	$O(\min(f_i))$	Intersection set

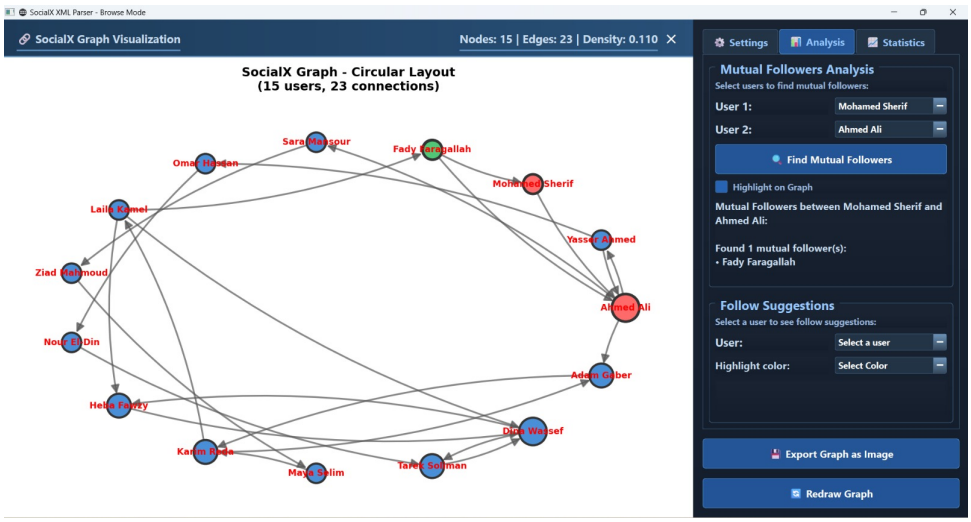

Figure 2.13: Mutual Followers (Nodes with green color)

Friend Suggestions

Function: `suggest_users_to_follow()`

Algorithm: Friends-of-friends recommendation with relevance scoring.

Listing 2.17: Friend Suggestion Algorithm

```
def suggest_users_to_follow(self, user_id, limit=5):
    following = set(self.G.successors(user_id))
    recommendations = defaultdict(int)

    for followed_user in following:
        for suggested in self.G.successors(followed_user):
            if suggested not in following and suggested != user_id
                :
                recommendations[suggested] += 1

    return sorted(recommendations.items(),
                  key=lambda x: x[1], reverse=True)[:limit]
```

Metric	Complexity	Notes
Time Complexity	$O(f \cdot g + r \log r)$	f = following, g = their following, r = results
Space Complexity	$O(r)$	Recommendation counts

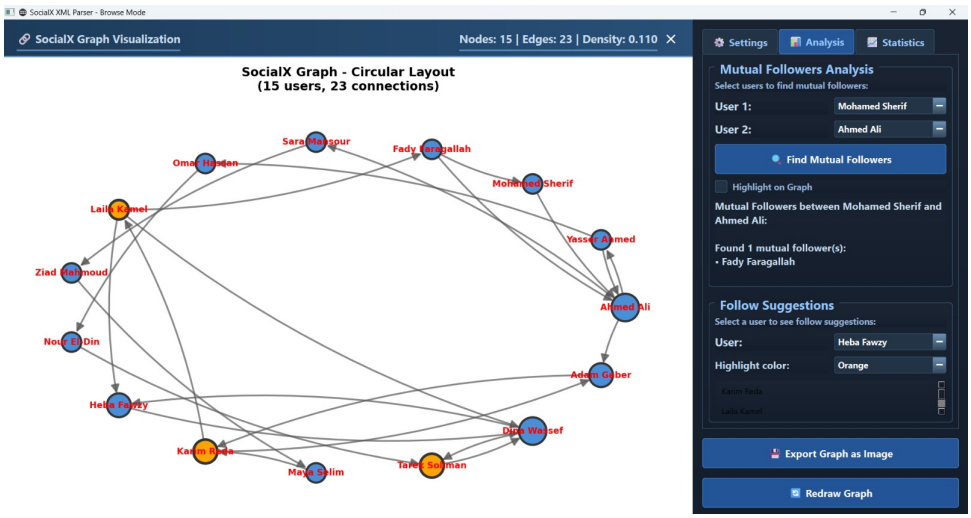

Figure 2.14: Friend Suggestions (Nodes with yellow color)

2.2.3 DataParser (data_parser.py)

Parses XML data into structured Python objects with caching.

User Parsing

Function: `parse_users()`
Algorithm: XML tree traversal with memoization.

Listing 2.18: `parse_users()` core implementation

```
def parse_users(self) -> Dict[str, User]:
    if self._users_cache is not None:
        return self._users_cache

    users_dict = {}
    users = self.xml_data.findall('.//user')
    for user_elem in users:
        user_id = self._extract_user_id(user_elem)
        if not user_id:
            continue
        name_elem = user_elem.find('name')
        user_name = name_elem.text.strip() if name_elem is not
            None and name_elem.text else f"User {user_id}"
        followers = self._extract_followers(user_elem)
        following = self._extract_following(user_elem)
        posts = len(user_elem.findall('.//post'))
        user = User(id=str(user_id), name=user_name, followers=
            followers, following=following, posts=posts)
        users_dict[str(user_id)] = user

    self._users_cache = users_dict
    return users_dict
```

Metric	Complexity	Notes
Time Complexity	$O(n)$	First call; $O(1)$ cached
Space Complexity	$O(u)$	User objects cache

2.2.4 ByteUtils (binary_utils.py)

Low-level binary packing utilities for compression.

Function	Time	Space
`pack_u16()`	$O(1)$	$O(1)$
`pack_u32()`	$O(1)$	$O(1)$
`unpack_u16()`	$O(1)$	$O(1)$
`unpack_u32()`	$O(1)$	$O(1)$

2.3 Complexity Summary Table

Table 2.1: Complete Complexity Analysis

Module	Function	Time	Space
XMLController	tokenize	$O(n)$	$O(n)$
XMLController	validate	$O(n \cdot m)$	$O(d)$
XMLController	autocorrect	$O(n \cdot d)$	$O(n)$
XMLController	format	$O(n)$	$O(n)$
XMLController	minify	$O(n)$	$O(n)$
XMLController	compress	$O(k \cdot n)$	$O(n)$
XMLController	decompress	$O(k \cdot n)$	$O(n)$
XMLController	export_to_json	$O(n)$	$O(u + p)$
GraphController	build_graph	$O(V + E)$	$O(V + E)$
GraphController	calculate_metrics	$O(V)$	$O(V)$
XMLTree	fromstring	$O(n \cdot d)$	$O(n)$
XMLTree	findall	$O(n)$	$O(d)$
NetworkAnalyzer	most_influential	$O(V)$	$O(V)$
NetworkAnalyzer	mutual_followers	$O(k \cdot f)$	$O(f)$
NetworkAnalyzer	suggest_users	$O(f \cdot g)$	$O(r)$

Data Flow

This chapter describes the complete lifecycle of data from user input through processing to output.

3.1 File Processing Pipeline

The data flows through the following stages:

1. **User Upload** (GUI/CLI Input)
2. **File I/O** (`file_io.read_file()`)
3. **Controller Processing** (`XMLController.set_xml_string()`)
4. **Operation Execution** (validate, format, compress, etc.)
5. **Output Generation** (`file_io.write_file()`)
6. **Output File** (saved to `output_samples/`)

3.2 Stage 1: User Input

3.2.1 GUI Mode

1. User launches application via `main.py`
2. Selects GUI mode (Option 1)
3. `AppManager` displays `LandingWindow`
4. User chooses:
   - **Browse Mode**: File picker dialog via `QFileDialog`
   - **Manual Mode**: Direct text input via `QTextEdit`

3.2.2 CLI Mode

1. User launches `python cli.py` or selects Option 2 from main menu
2. REPL displays bash-style prompt: `~/path $`
3. User enters command with arguments: `format -i input.xml -o output.xml`
4. `argparse` parses arguments into `args` namespace

3.3 Stage 2: File Reading

The `file_io.py` module provides a safe abstraction for disk operations:

Listing 3.1: Safe File Reading

```
def read_file(path: str) -> Tuple[bool, str]:
    """
    Returns (success, content or error message).
    Avoids exceptions leaking into controllers.
    """
    try:
        content = Path(path).read_text(encoding="utf-8")
        return True, content
    except Exception as e:
        return False, str(e)
```

Key Design Decision: Controllers never touch disk operations directly. This allows:

- Consistent error handling across GUI and CLI
- Easy mocking for unit tests
- Centralized encoding management (UTF-8)

3.4 Stage 3: Processing

3.4.1 Controller Initialization

```
editor = XMLController()
editor.set_xml_string(file_content)
```

3.4.2 Operation Execution

Each operation follows the same pattern:

1. Tokenize XML string into structured tokens
2. Apply operation-specific algorithm
3. Return processed string or data structure

Example: Format Operation

```
def format(self) -> str:
    tokens = self._get_tokens()       # Step 1: Tokenize
    formatted = []
    level = 0
    for token in tokens:              # Step 2: Process
        if token.startswith('</'):
            level -= 1
            formatted.append(indent * level + token)
```

```
        elif token.startswith('<'):
            formatted.append(indent * level + token)
            level += 1
        else:
            formatted.append(indent * level + token)
    return '\n'.join(formatted)       # Step 3: Return
```

3.5 Stage 4: Output Generation

3.5.1 File Output

Listing 3.2: Safe File Writing

```
def write_file(path: str, data: str) -> Tuple[bool, str]:
    try:
        formatted = pretty_format(data)
        with open(path, "w", encoding="utf-8") as file:
            file.write(formatted)
        return True, "XML file written successfully."
    except OSError as e:
        return False, f"File error: {e}"
```

3.5.2 GUI Display

For GUI mode, results are displayed in `QTextEdit` widgets with syntax highlighting.

3.5.3 CLI Output

For CLI mode, results are printed to stdout with colorama formatting:

```
print(f"{Fore.GREEN}Saved to {args.output}{Style.RESET_ALL}")
```

3.6 Graph Data Flow

For network analysis operations, an additional flow branch exists:

1. `GraphController.set_xml_data(content)`
2. `DataParser` extracts nodes and edges
3. `NetworkX DiGraph` constructed
4. `NetworkAnalyzer` performs analysis
5. Results returned as Python dictionaries

Technologies Used

4.1 Core Libraries

Table 4.1: Technology Stack

Library	Version	Justification
PySide6	≥6.0.0	Qt bindings for Python. Chosen over PyQt6 for LGPL licensing. Provides modern widgets, signals/slots architecture, and cross-platform support.
NetworkX	≥2.6.0	Industry-standard graph library. Provides DiGraph for directed social networks, built-in algorithms for centrality, paths, and clustering.
NumPy	≥1.20.0	Efficient numerical operations for network metrics aggregation. Used for mean/max calculations on degree distributions.
Matplotlib	≥3.5.0	Graph visualization backend. Integrates with NetworkX for drawing social network graphs with customizable layouts.
Colorama	any	Cross-platform terminal colors. Enables colored CLI output on Windows/Linux/macOS without ANSI escape code issues.
Base64	stdlib	Binary-to-text encoding for compressed data. Ensures compressed output is safe for file storage and UI display.

4.2 Development Tools

Tool	Purpose
Python 3.8+	Runtime environment
argparse	CLI argument parsing
shlex	Shell-style argument splitting
re	Regular expression matching
textwrap	Text wrapping for formatting
dataclasses	Structured data objects
pathlib	Modern file path handling
PyInstaller	Executable packaging

4.3 Build System

The project includes `Builder.bat` for Windows executable generation:

Listing 4.1: Build Script

```
@echo off
echo Installing Dependencies...
pip install PySide6 networkx matplotlib numpy colorama pyinstaller

echo Building Executable...
pyinstaller --name "SocialX" --onefile --clean \
            --icon "assets\icon.ico" \
            --add-data "assets\icon.ico;." main.py

echo Build Successful!
echo Executable: dist\SocialX.exe
```

Conclusion

The SocialX XML Editor demonstrates a well-architected software system with:

- **Clean Separation**: UI code is completely decoupled from business logic
- **Dual Interface**: Same controllers work for both GUI and CLI
- **Efficient Algorithms**: Documented O(n) tokenization, O(V+E) graph operations
- **Modern Patterns**: MVC-like architecture, signal/slot communication
- **Comprehensive Features**: Validation, formatting, compression, network analysis

Demonstration Video:

https://www.youtube.com/watch?v=Ld3K32fZ0R4&t=2s