
# Usman Institute of Technology University

## Department of Computer Science

### Spring 2025

## CSC205 – Theory of Automata
## Project Report
## DFA Roman Number Validator

| | |
|---|---|
| Student Name | Ahmad Naaem Saad |
| Student ID | 23SP-006-CS |
| Section | CS(a) |
| Semester | V |

# Table of Contents

## Contents

# Table of Figures

# Objective

The objective of this project is to implement a **Deterministic Finite Automaton (DFA)** that validates **Roman numerals ranging from I (1) to L (50)**. The DFA operates by transitioning through predefined states based on user input, checking whether the sequence adheres to Roman numeral formation rules.

The project also includes:

- A **Graphical User Interface (GUI)** built using **Tkinter**, where users can enter Roman numerals.
- A **visual transition diagram** generated with **NetworkX and Matplotlib**, showing real-time traversal of states based on user input.
- Feedback indicating whether the numeral is valid and, if valid, displaying its decimal equivalent.

This project showcases how theoretical automata concepts can be translated into interactive software for a practical validation problem.

# Problem Statement

Roman numerals are non-positional and follow strict syntactic conventions:

- Certain characters (like `I`, `X`, `C`) can be repeated, while others (`V`, `L`, `D`) cannot.
- Subtractive notation is used (e.g., `IV` for 4, `IX` for 9).
- Characters must appear in a decreasing order of value unless a valid subtractive combination is used.

These rules make validation non-trivial and prone to user error. Traditional string checks can be complex and error-prone, especially if they try to encode every Roman numeral manually.

**Automata theory**, specifically **Deterministic Finite Automata (DFA)**, provides a clean and efficient solution:

- States and transitions can encode legal Roman numeral patterns.
- Invalid transitions are handled through a **dead state**, which automatically rejects incorrect input.
- Once the input is fully processed, the DFA determines validity based on the current state being accepting or not.

This project highlights how theoretical models can elegantly solve practical problems like pattern recognition and input validation.

# Applied Concepts

The central theoretical foundation of the project is the **Deterministic Finite Automaton (DFA)**, one of the core concepts in automata theory.

## Key Concepts Implemented:

- **Deterministic Transitions**: For every state and input symbol, there is exactly one next state. This eliminates ambiguity and ensures predictable behavior.
- **Accepting States**: These represent valid end conditions (i.e., well-formed Roman numerals).
- **Dead State**: Any invalid sequence leads to this non-accepting state, ensuring incorrect numerals are rejected.
- **Alphabet**: The input characters allowed are `I`, `V`, `X`, and `L`, which cover the range from 1 to 50.
- **Limit**: Since DFA are deterministic and are finite, we limit our DFA to range of 50.

This project deliberately focuses on DFA because the Roman numeral validation problem—within a finite bounded range—is well-suited for finite-state machines without requiring memory (stack/tape).

**Visual of DFA being used to validate:**
To note:

- The following DFA doesn't have a dead state for the sake of ease of understanding.
- All green states are considered as final states

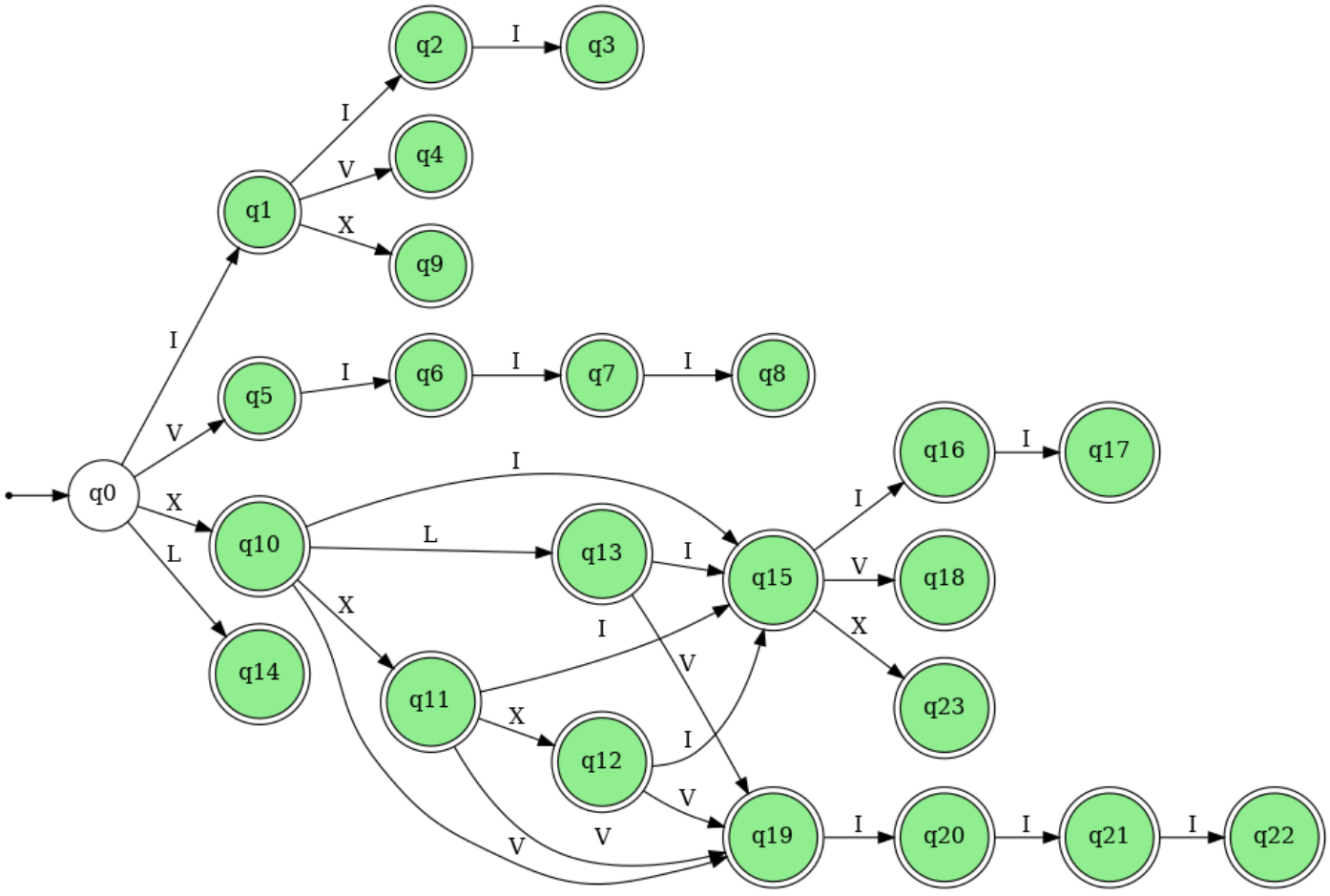


*Figure 1:DFA Visual (without dead state)*

# Methodology and Logic

The solution architecture combines **Python programming**, **graph theory**, and **GUI design** to bring the DFA to life in an interactive format.

## 1. State Definition

- Each DFA state corresponds to a valid prefix of a Roman numeral.
- The starting state `q0` branches into valid initial characters (`I`, `V`, `X`, `L`).
- Additional states handle valid continuation sequences based on Roman numeral rules.

## 2. Transition Logic

- Transitions are defined in a dictionary format: `transitions[state][symbol] = next_state`.
- Any missing or invalid transitions are routed to a **dead state (`q_dead`)**.
- This ensures total function coverage: every (state, symbol) pair is accounted for.

## 3. Visualization

- **NetworkX** is used to generate a directed graph of states and transitions.
- States are color-coded:
  - Blue arrows for valid transitions
  - Red for dead state
  - Green nodes for accepting states
- **Matplotlib** renders the graph inside the Tkinter window.

## 4. User Interface

- Built using **Tkinter** with simple input and output components:
  - Entry field for Roman numeral
  - Button to validate input
  - Label for displaying result (valid/invalid, decimal value)
  - Canvas showing the DFA graph

## 5. Validation Process

- As input is processed character-by-character, the DFA traces a path through its states.
- This path is highlighted dynamically on the graph.
- If the end state is in the set of accepting states, the input is considered valid.

## 6. Nodes made by matplot and networkx:

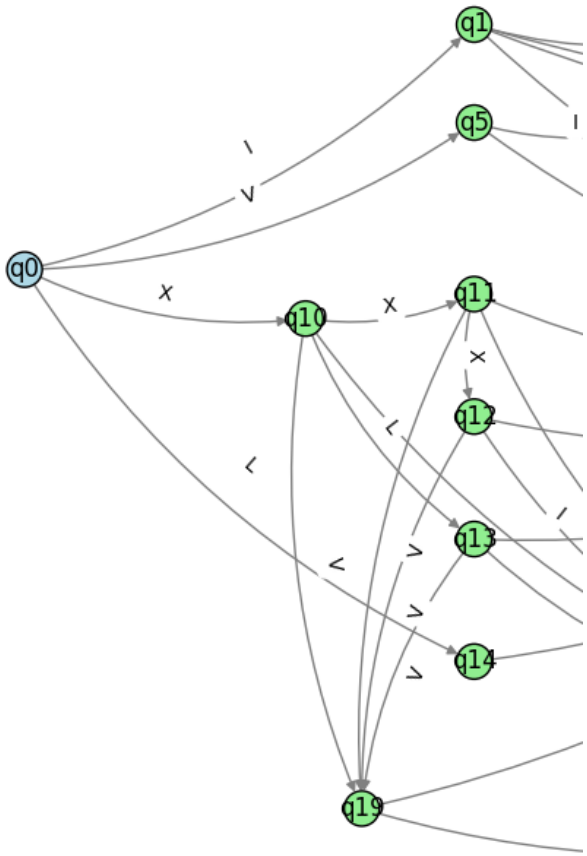


*Figure 2: Node use example*

# Code Working:

Key elements of the code include:
- A dictionary (`transitions`) encoding state changes
- A `draw_dfa()` function to visually show the DFA and highlight valid/invalid paths
- Real-time user feedback using labels for valid/invalid numerals

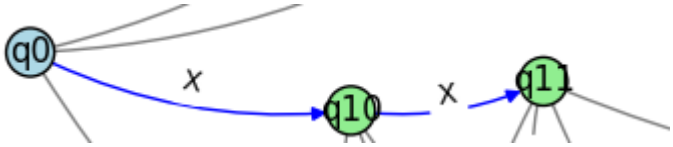


*Figure 3: Highlight mechanism*

# Classes and Functions Overview

The code for the Roman numeral DFA validator is organized into modular components using object-oriented programming (OOP) principles and standard Python libraries for GUI development and graph visualization.

## Main Class: RomanDFAApp

This is a subclass of `tk.Tk`, which serves as the root window of the application. It integrates all user interface components, DFA logic, and visualization.

## Constructor: __init__(self)

- Initializes the main GUI window with an appropriate title and size.
- Sets up input fields (`Entry`), buttons (`Button`), and result display (`Label`) using Tkinter.
- Creates a Matplotlib figure and embeds it into the Tkinter window using `FigureCanvasTkAgg`.
- Builds the DFA graph and renders its initial structure.

```
def __init__(self):
    super().__init__()
    self.title("Clean Roman Numeral DFA Validator")
    self.geometry("1600x900")
    self.protocol("WM_DELETE_WINDOW", self.on_close)

    tk.Label(self, text="Enter Roman Numeral (1 to 50):", font=("Arial", 14)).pack(pady=10)
    self.entry = tk.Entry(self, font=("Arial", 14), width=20, justify='center')
    self.entry.pack()
    tk.Button(self, text="Validate & Show Transitions", font=("Arial", 12), command=self.validate).pack(pady=10)
    self.result = tk.Label(self, text="", font=("Arial", 14))
    self.result.pack(pady=10)

    self.figure, self.ax = plt.subplots(figsize=(20, 12))
    self.canvas = FigureCanvasTkAgg(self.figure, master=self)
    self.canvas.get_tk_widget().pack()

    self.dfa_graph = self.build_dfa_graph()
    self.draw_dfa()
```

*Figure 4: Constructor*

## Method: on_close(self)

- Handles proper closure of the GUI window by destroying the root window and ending the application.

```
def on_close(self):
    self.destroy()
    self.quit()
```

*Figure 5: on_close()*

## Method: roman_to_decimal(self, s)

- Converts a valid Roman numeral to its decimal value using a dictionary-based mapping.
- Traverses the input string from right to left and applies subtraction logic where appropriate (e.g., IV = 4, not 6).

```
def roman_to_decimal(self, s):
    roman_map = {'I': 1, 'V': 5, 'X': 10, 'L': 50}
    total = 0
    prev = 0
    for ch in reversed(s):
        value = roman_map[ch]
        if value < prev:
            total -= value
        else:
            total += value
        prev = value
    return total
```

*Figure 6: roman_to_decimal()*

## Method: build_dfa_graph(self)

- Constructs a directed graph using NetworkX based on the predefined transition rules.
- Adds edges between states and labels them with corresponding input characters.

```
def build_dfa_graph(self):
    G = nx.DiGraph()
    edge_labels = {}
    for state, edges in transitions.items():
        for symbol, target in edges.items():
            G.add_edge(state, target)
            key = (state, target)
            if key in edge_labels:
                edge_labels[key] += f",{symbol}"
            else:
                edge_labels[key] = symbol
    nx.set_edge_attributes(G, edge_labels, "label")
    return G
```

*Figure 7: build_dfa_graph*

## Method: draw_dfa(self, highlight_path=None)

- Draws the DFA on the embedded Matplotlib canvas.
- Applies different colors to nodes based on their type:
  - Light blue for normal states
  - Light green for accepting states
  - Red for the dead state
- Highlights the path taken by a valid input if provided.

```
def draw_dfa(self, highlight_path=None):
    self.ax.clear()
    pos = {k: (v[0]*400, v[1]*400) for k, v in state_positions.items()}
    node_colors = ['red' if n == 'q_dead' else 'lightgreen' if n in accepting_states else 'lightblue' for n in self.dfa_graph.nodes]

    nx.draw_networkx_nodes(self.dfa_graph, pos, ax=self.ax, node_color=node_colors, edgecolors='black')
    nx.draw_networkx_labels(self.dfa_graph, pos, ax=self.ax)

    edge_colors = ['blue' if highlight_path and (u, v) in highlight_path else 'gray' for u, v in self.dfa_graph.edges()]
    nx.draw_networkx_edges(self.dfa_graph, pos, ax=self.ax, edge_color=edge_colors, connectionstyle='arc3,rad=0.15')

    labels = {(u, v): d["label"] for u, v, d in self.dfa_graph.edges(data=True)}
    nx.draw_networkx_edge_labels(self.dfa_graph, pos, edge_labels=labels, ax=self.ax)

    self.ax.set_title("DFA for Roman Numerals (I to L)")
    self.ax.axis("off")
    self.canvas.draw()
```

*Figure 8: draw_dfa()*

## Method: validate(self)

- Processes user input from the GUI.
- Traverses the DFA by reading each character and updating the state accordingly.
- Records the path taken and checks if the final state is an accepting state.
- Displays whether the Roman numeral is valid or not and shows its decimal equivalent if valid.
- Updates the graph to visually reflect the path.

```
def validate(self):
    input_text = self.entry.get().strip().upper()
    state = 'q0'
    path = []
    for ch in input_text:
        if ch not in alphabet:
            self.result.config(text=f"❌ Invalid character: {ch}", fg="red")
            return
        next_state = transitions[state][ch]
        path.append((state, next_state))
        state = next_state
```

*Figure 9: validate()*

## Global Definitions (Outside the Class)

These are defined outside of the class and serve as configuration data for the DFA:

- `state_positions`: A dictionary that assigns (x, y) coordinates to each state for graph layout.

```
state_positions = {
    # Start state
    'q0': (-2, 6),

    # Top layer (q1 to q4)
    'q1': (2, 16), 'q2': (4.5, 16), 'q3': (7.5, 16), 'q4': (11, 16),

    # Middle layer (q5 to q9)
    'q5': (2, 12), 'q6': (3.8, 12), 'q7': (6.5, 12), 'q8': (9.5, 12), 'q9': (12.5, 12),

    # Bottom-left vertical stack (q10, q11, q12, q13, q14)
    'q10': (0.5, 4), 'q11': (2, 5), 'q12': (2, 0), 'q13': (2, -5), 'q14': (2, -10),

    # Bottom horizontal (q15 to q18)
    'q15': (5, -12), 'q16': (8, -12), 'q17': (11, -12), 'q18': (14, -12),

    # Small vertical segment (q19, q20, q21, q22)
    'q19': (1, -16), 'q20': (6.5, -16), 'q21': (9.5, -16), 'q22': (12, -16),

    # Dead state
    'q_dead': (8, 2)
}
```

*Figure 10: state_positions*

- `accepting_states`: A set of states where input is considered valid and accepted.

```
accepting_states = {
    'q1', 'q2', 'q3', 'q4', 'q5', 'q6', 'q7', 'q8', 'q9', 'q10', 'q11', 'q12',
    'q13', 'q14', 'q15', 'q16', 'q17', 'q18', 'q19', 'q20', 'q21', 'q22'
}
```

*Figure 11: accepting_states*

- `alphabet`: The allowed Roman numeral symbols in this system: `['I', 'V', 'X', 'L']`.

```
alphabet = ['I', 'V', 'X', 'L']
```

*Figure 12: alphabet*

- `transitions`: A nested dictionary where each key is a current state, and each value maps input symbols to next states.

```
transitions = {
    'q0': {'I': 'q1', 'V': 'q5', 'X': 'q10', 'L': 'q14'},
    'q1': {'I': 'q2', 'V': 'q4', 'X': 'q9'},
    'q2': {'I': 'q3'},
    'q5': {'I': 'q6'},
    'q6': {'I': 'q7'},
    'q7': {'I': 'q8'},
    'q10': {'X': 'q11', 'L': 'q13', 'I': 'q15', 'V': 'q19'},
    'q11': {'X': 'q12', 'I': 'q15', 'V': 'q19'},
    'q12': {'I': 'q15', 'V': 'q19'},
    'q13': {'I': 'q15', 'V': 'q19'},
    'q15': {'I': 'q16', 'V': 'q18', 'X': 'q18'},
    'q16': {'I': 'q17', 'X': 'q18'},
    'q19': {'I': 'q20'},
    'q20': {'I': 'q21'},
    'q21': {'I': 'q22'},
}
```

*Figure 13: transitions*

- Additional logic ensures every state has defined transitions for all symbols, defaulting to a dead state (`q_dead`) where necessary.

```
# Add missing transitions to dead state
all_states = set(state_positions)
for state in all_states:
    transitions.setdefault(state, {})
    for symbol in alphabet:
        if symbol not in transitions[state]:
            transitions[state][symbol] = 'q_dead'
transitions['q_dead'] = {s: 'q_dead' for s in alphabet}
```

*Figure 14: complete transitions for dead state*

```
import tkinter as tk
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
```

*Figure 15: Imports*

## Theoretical Concept Mapping

This project is a direct application of **Deterministic Finite Automata (DFA)** from automata theory to the validation of Roman numerals. Each core concept from the theory is mapped carefully to a programming construct and visual representation.

**States**

- In the DFA, each state represents a valid prefix of a Roman numeral between I and L (1–50).
- The start state is q0, from which all processing begins.
- States q1 to q22 represent different valid intermediate or final patterns (e.g., after one or two Is, a V, or a valid sequence like XL).
- These states are not arbitrary; they reflect Roman numeral composition logic:
    - Repetition of I up to three times (e.g., I, II, III)
    - Proper order of subtraction (e.g., IV, IX, XL)
    - No invalid combinations like IIII, VV, or VX

**Transitions**

- Transitions follow deterministic rules: for each (state, input) pair, there is exactly one valid next state.
- For example:
    - From q0, if the input is I, the DFA transitions to q1.
    - From q1, input I leads to q2, while input V leads to q4 (IV).
- The full transition dictionary (transitions) ensures coverage of all valid scenarios. Any missing or invalid route leads to a dead state.

**Accepting States**

- Accepting states include all the final forms of valid Roman numerals (e.g., q1, q4, q8, q17, etc.).
- After processing the full input, if the DFA ends in an accepting state, the numeral is declared valid.
- The accepting_states set is explicitly defined in the code to represent these end conditions.

**Dead State**

- If at any point an invalid character is entered, or an invalid sequence is formed (e.g., IIII or VX), the DFA transitions to a special state: q_dead.
- q_dead is a trap state with self-loops on all inputs, ensuring that no valid transition is possible afterward.
- This models how DFA handles unrecognized patterns—once in a dead state, the input is rejected.

## Determinism

- The entire design adheres to deterministic behavior:
  - Each state has only one defined transition for each character in the alphabet (I, V, X, L).
  - There is no ambiguity or need for backtracking.

## Graphical Representation

- The DFA is visualized using NetworkX and Matplotlib.
- Each node (state) is placed in a 2D space using state_positions.
- Edges represent transitions and are labeled with the corresponding input character.
- Colors indicate state types:
  - Green: Accepting
  - Red: Dead state
  - Blue edges: Valid transition paths

This visual representation not only aids debugging but also serves as an educational tool to understand DFA structure and flow.

## Full DFA Visual in Code UI:

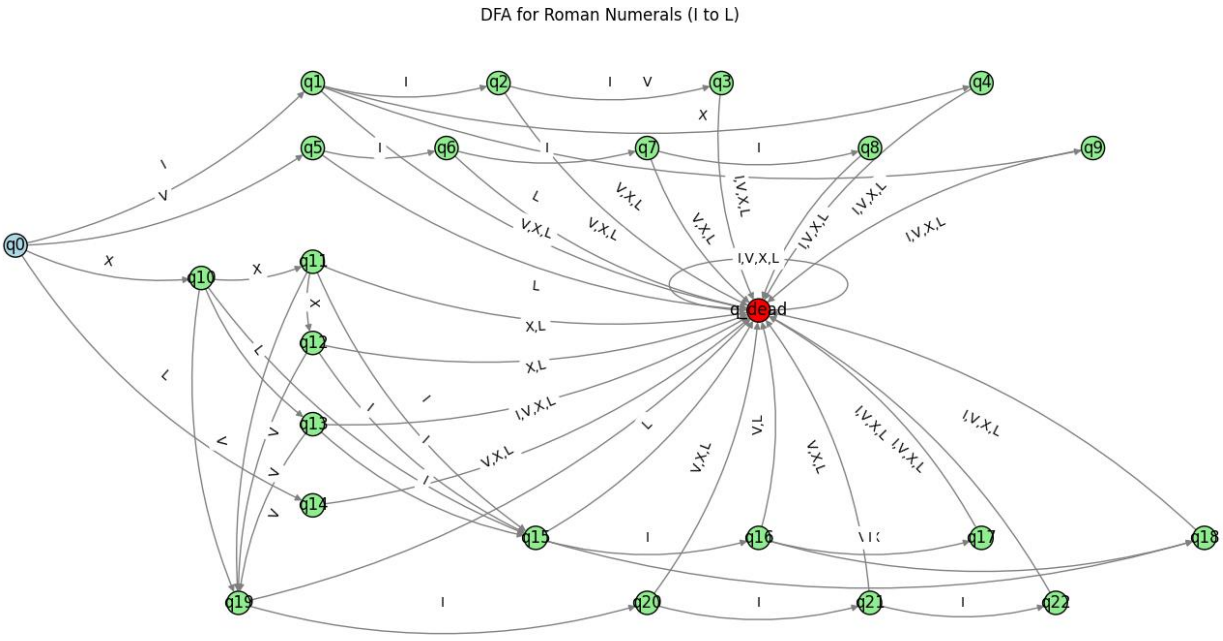


*Figure 16:Full DFA in code UI*

# Screenshots of Simulation or Output

## Front Screen:

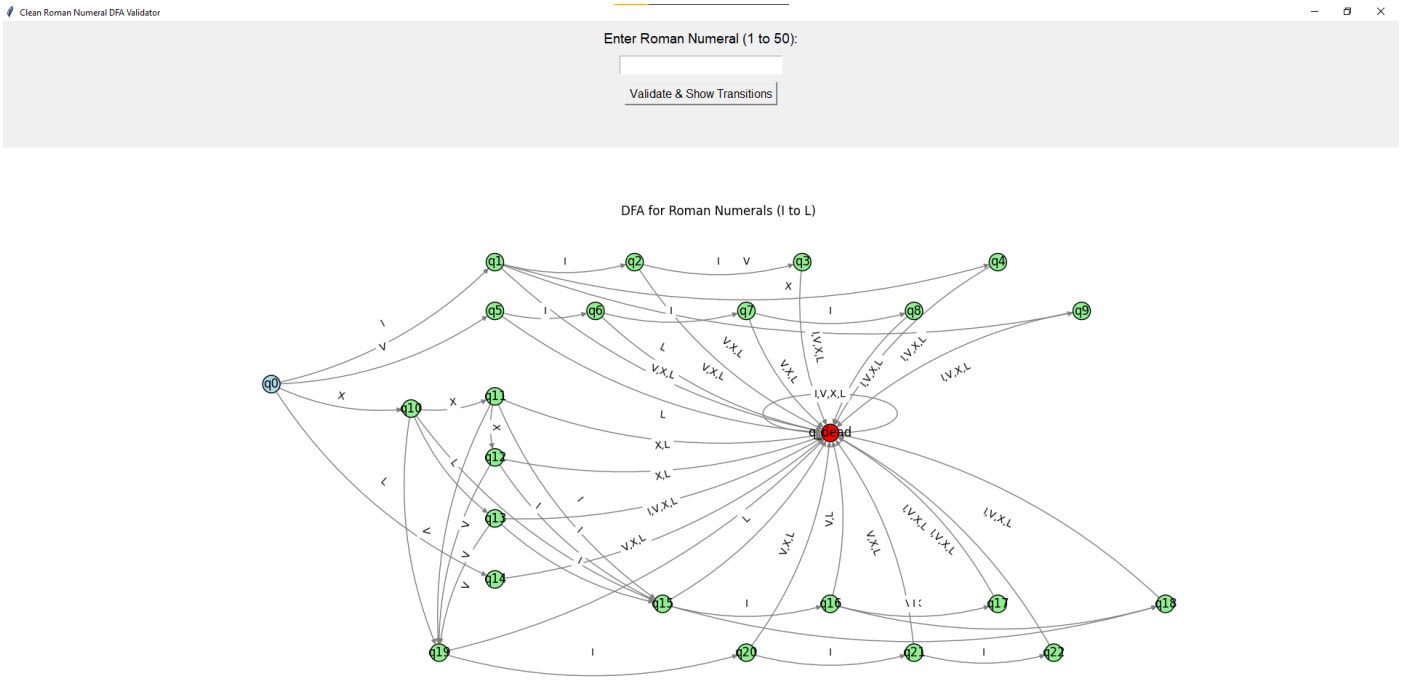


*Figure 17: Front Screen*

## Valid Roman Numeral Screen:

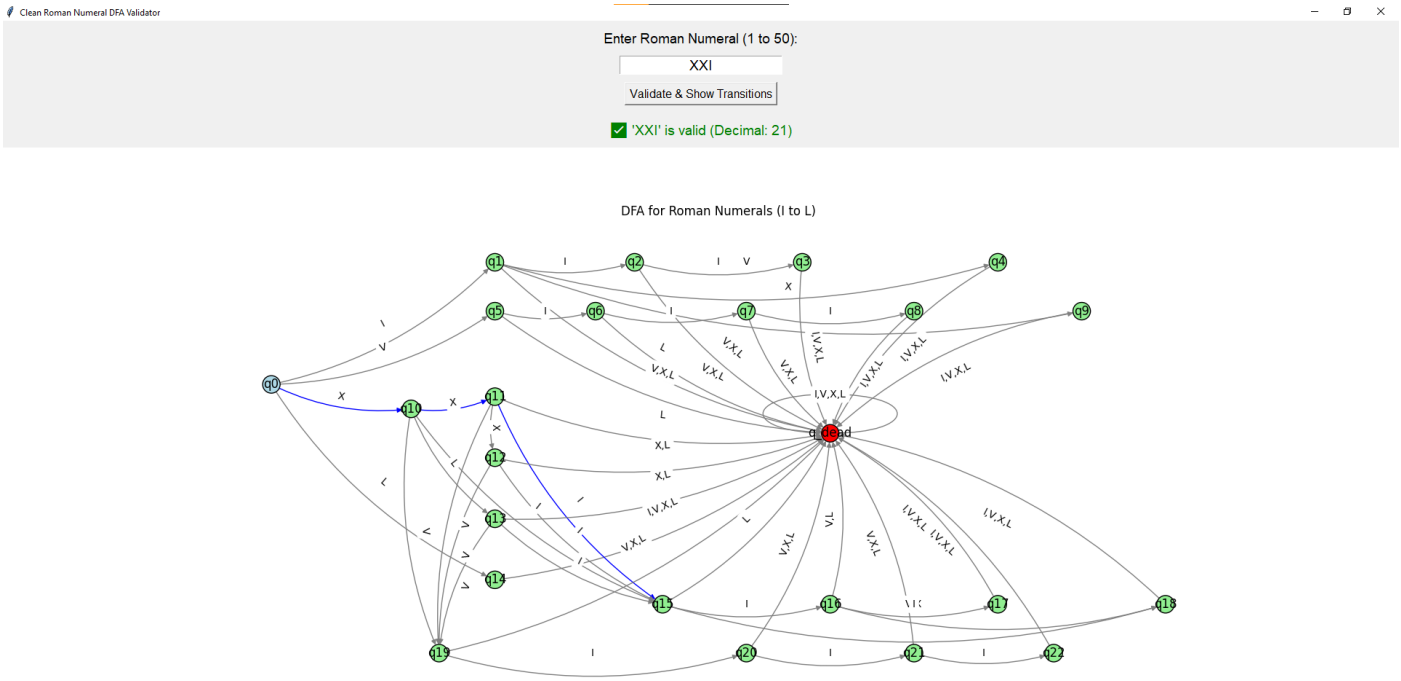


*Figure 18: Valid Screen*

# Non-Valid Roman Numeral Screen:

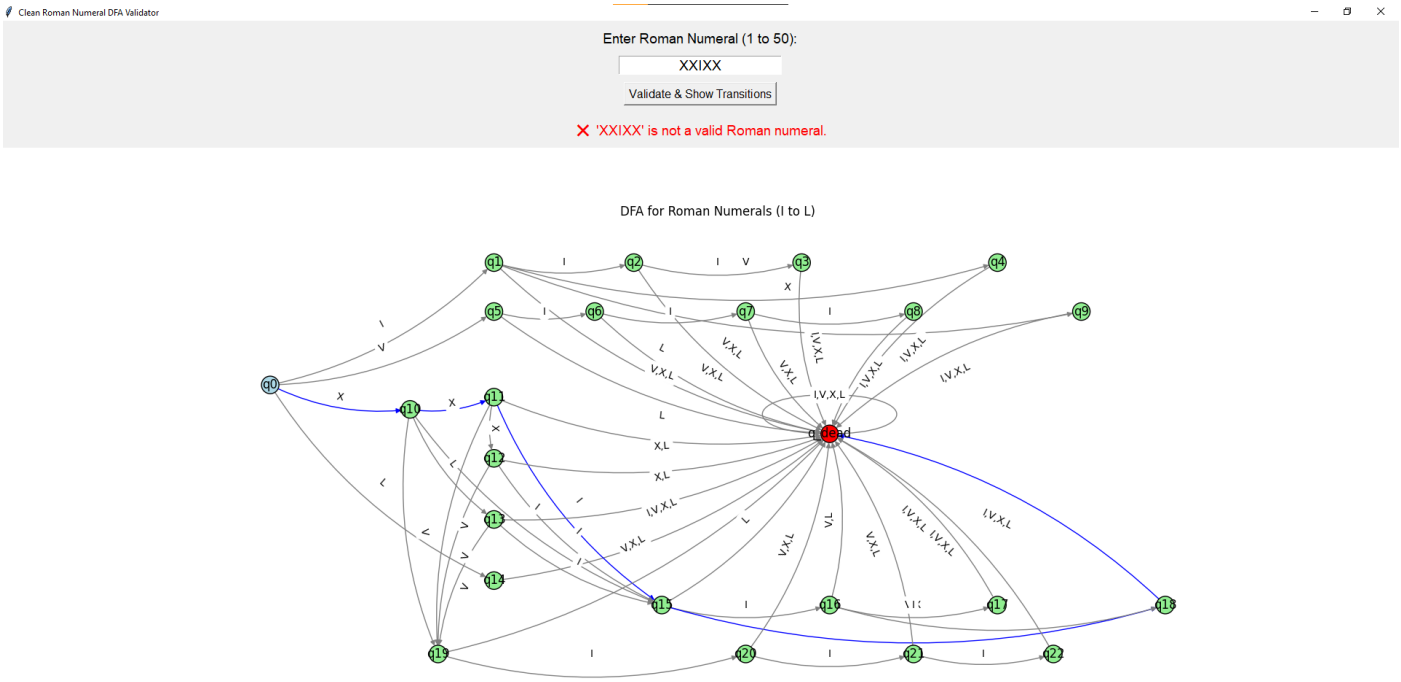


*Figure 19: Non-Valid Screen*

# Roman to Decimal UI:

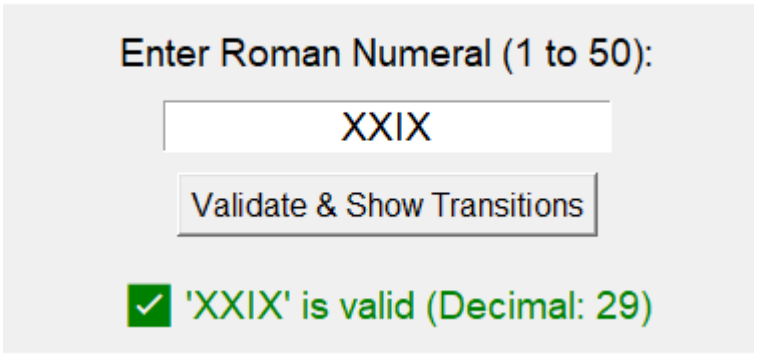


*Figure 20: Conversion UI*

## Challenges and Resolutions

During the development of this DFA-based Roman numeral validator, several challenges were encountered both conceptually and technically. These challenges provided valuable learning opportunities, especially in translating abstract automata theory into a practical and interactive implementation.

**Challenges Faced:**

1. **Managing transition rules for all Roman numeral combinations up to L (50):**
    - Roman numerals involve both additive and subtractive patterns, and ensuring valid sequences like XL, IX, XV, and avoiding invalid ones like IIII or VV required a thorough mapping of states and transitions.
    - Covering all edge cases was difficult, especially since each new rule introduced more potential combinations.
2. **Designing a DFA structure that is organized and not visually overwhelming:**
    - Initially, the DFA became very large and confusing when visualized.
    - With 20+ states and dozens of transitions, positioning them in a logical and readable layout was difficult.
    - Avoiding clutter while still maintaining accurate visual representation of transitions required experimentation with spacing and groupings.
3. **Ensuring that the GUI remained responsive and user-friendly:**
    - Integrating graph rendering using NetworkX and Matplotlib within a Tkinter window created performance concerns.
    - Careful placement of widgets and efficient redrawing of the graph on every input was necessary to maintain a smooth user experience.
4. **Dynamic graph highlighting based on user input:**
    - The DFA needed to visually trace the state path taken by the input.
    - Highlighting only the relevant edges and differentiating them from the rest required custom logic to manage color changes dynamically.

**Resolutions:**

- A **comprehensive transition map** was developed by manually analyzing all valid Roman numeral structures from 1 to 50.
- **Dead state routing** was used to simplify the DFA's logic. Any undefined transitions immediately lead to a single dead state, which eliminates the need for multiple error-handling branches.
- To **organize the DFA graph visually**, each group of states was placed in layers (top, middle, bottom) based on their logic roles—this reduced confusion and made the visual flow easier to follow.
- **Graph highlighting** was implemented by tracking the path of transitions and recoloring only those edges using NetworkX's edge drawing methods

## Conclusion and Observations

This project demonstrates how deterministic finite automata, a fundamental concept in automata theory, can be applied to solve a real-world validation problem—Roman numeral verification.

Through this project, the theoretical constructs of states, transitions, accepting conditions, and dead states were not only implemented but also made interactive and visual. The use of a GUI and real-time graph visualization helped bring abstract concepts to life.

Some key takeaways and observations include:

- DFA is a suitable model for pattern-based input validation, especially when the input domain is finite and follows well-defined syntactic rules.
- The balance between **functionality and usability** (both in logic and UI) is crucial for practical applications.
- Visualizing a large DFA is not just helpful for debugging—it becomes a valuable educational and explanatory tool.
- Organizing DFA states into a structured layout with proper spacing greatly improves readability and usability.

**Future Enhancements:**

- Support Roman numerals beyond L (50), up to C (100), D (500), or even M (1000), which would require a more complex DFA or possibly a PDA.
- Convert the application into a **web-based tool** using JavaScript and D3.js for visualization or deploy the current Python version using a web framework like Flask.
- Add real-time suggestions or explanations for invalid inputs to improve learning for users.

This project not only reinforced theoretical knowledge but also developed practical skills in software design, user interface development, and data visualization.