ATRIA INSTITUTE OF TECHNOLOGY

Developing a Complete Professional

(Approved by AICTE, Affiliated to VTU and NAAC Accredited)

Department of Computer Science & Engineering

ACTIVITY CONDUCTED REPORT

Activity Name:	**Practical Approaches of Computation Theory**
Co-ordinator:	**Prof M. Niranjani**
Date:	**06-12-2024**
Duration:	**10:00 am - 12:45 pm**

Team Members –
Sharon A Dobbin (1AT22CS120)
Pooja Vijay Bijapur (1AT23CS404)
Zainab Bi (1AT22CS118)
Sneha C (1AT22CS099)

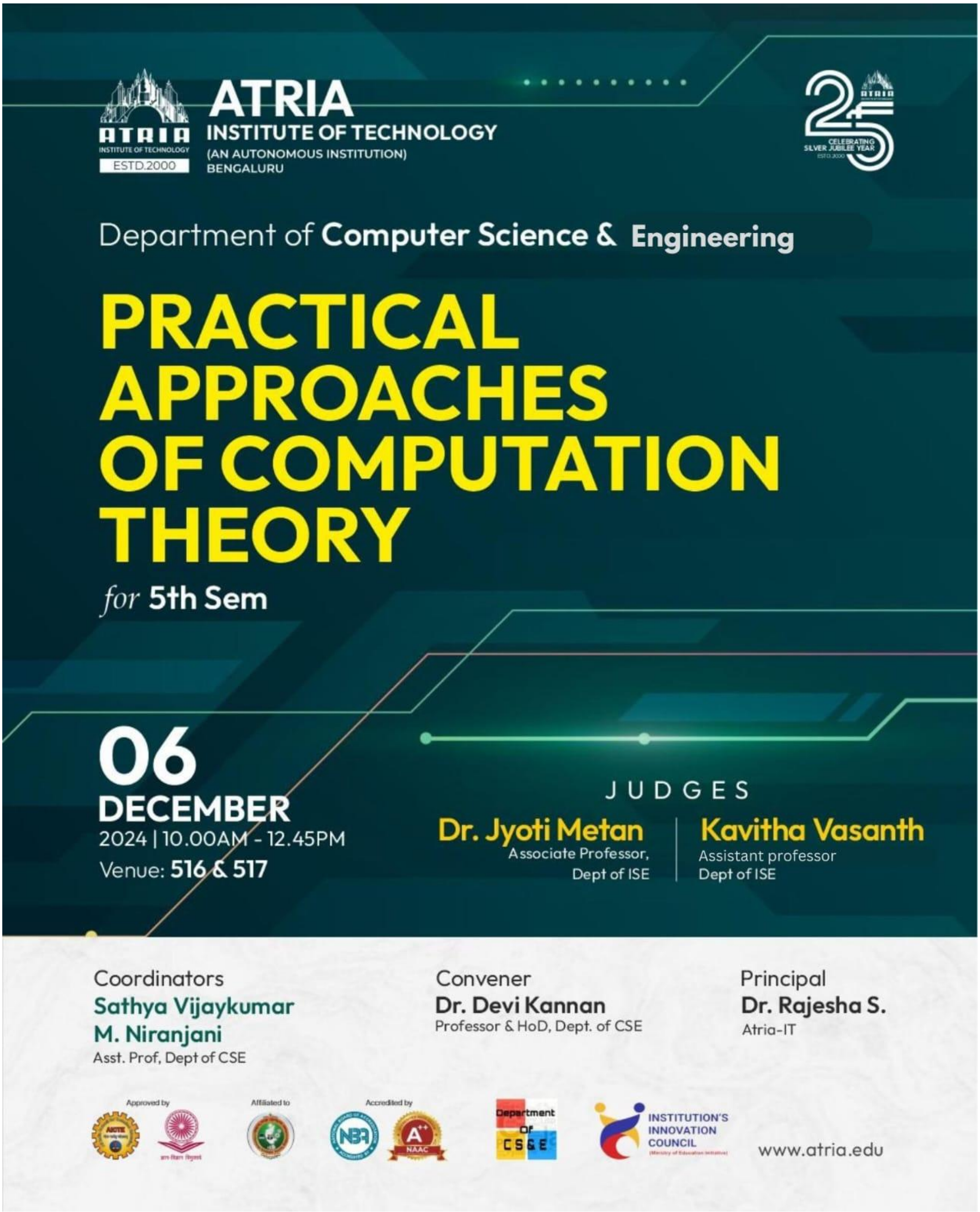
ATRIA INSTITUTE OF TECHNOLOGY

Developing a Complete Professional

(Approved by AICTE, Affiliated to VTU and NAAC Accredited)

ACADEMIC YEAR 2024-25
TABLE OF CONTENTS

BROCHURE/INVITATION



ATRIA
INSTITUTE OF TECHNOLOGY
(AN AUTONOMOUS INSTITUTION)
BENGALURU

ESTD.2000

CELEBRATING SILVER JUBILEE YEAR

Department of **Computer Science & Engineering**

PRACTICAL APPROACHES OF COMPUTATION THEORY

for **5th Sem**

06 DECEMBER
2024 | 10.00AM - 12.45PM
Venue: **516 & 517**

JUDGES

Dr. Jyoti Metan
Associate Professor,
Dept of ISE

Kavitha Vasanth
Assistant professor
Dept of ISE

Coordinators
Sathya Vijaykumar
M. Niranjani
Asst. Prof, Dept of CSE

Convener
Dr. Devi Kannan
Professor & HoD, Dept. of CSE

Principal
Dr. Rajesha S.
Atria-IT

Approved by Affiliated to Accredited by

Department of CS&E

INSTITUTION'S INNOVATION COUNCIL

www.atria.edu

ATRIA INSTITUTE OF TECHNOLOGY

Developing a Complete Professional

(Approved by AICTE, Affiliated to VTU and NAAC Accredited)

PROGRAM SCHEDULE

Department of Computer Science & Engineering

TIME	EVENTS
10:00 AM – 12:45 PM	PRACTICAL APPROACHES OF COMPUTATION THEORY.

ATRIA INSTITUTE OF TECHNOLOGY

Developing a Complete Professional

(Approved by AICTE, Affiliated to VTU and NAAC Accredited)

SESSION REPORT

SESSION REPORT			
Session Name	PRACTICAL APPROACHES OF COMPUTATION THEORY		
Resource Person	DR JYOTI METAN AND PROF KAVITHA VASANTH		
Coordinator	PROF SATHYA VIJAYKUMAR, PROF M. NIRANJANI		
Date	06-12-2024		10:00 AM – 12:45 PM
Intended Participants	Students of 5th Sem CSE		

Practical Approaches to Computation Theory – Theory of Computation (BCS503) Project Expo

Project Focus: This project centers on simulating the conversion of a **Regular Expression (RE)** into a **Non-deterministic Finite Automaton (NFA)** with epsilon (ε) transitions, using **Thompson's Construction Algorithm**. The project's primary objective was to provide an educational and interactive platform to demonstrate this conversion process visually. By utilizing Python and its rich ecosystem of libraries, the project effectively bridges theoretical concepts and practical implementation.

Key Highlights and Features

1. **Python Programming:**
    - **Core Implementation:**
        - The project implements Thompson's Construction Algorithm, a systematic method to construct an NFA from a regular expression.
        - It supports fundamental operations like:
            - **Concatenation** (.): Linking two patterns sequentially.
            - **Union** (|): Allowing a choice between two patterns.
            - **Kleene Star** (*): Enabling zero or more repetitions of a pattern.
    - **State Management:**
        - Each state is represented as an object containing its transitions and acceptance conditions.
2. **Graphviz for Visualization:**
    - **Graph Construction:**
        - Used to visually depict the NFA as a graph with states (nodes) and transitions (edges).
        - Includes epsilon (ε) transitions, which are a special feature of NFAs.
    - **Aesthetic Output:**
        - States are styled to distinguish between initial, final, and intermediate states (e.g., using colours and shapes).
        - Horizontal layouts improve clarity of visualization.
    - **Export Capability:**

        - The final NFA graph is rendered and saved as a high-resolution image (PNG format).

3. **Python Pillow (PIL) for Image Processing:**
    - **Dynamic Image Handling:**
        - The NFA image is loaded and resized dynamically based on the dimensions of the GUI window.
        - Ensures a seamless display experience, regardless of screen size.
4. **Tkinter for GUI:**
    - **Interactive User Interface:**
        - The GUI allows users to input a regular expression and view both the terminal output (for debugging or success messages) and the graphical visualization of the NFA.
    - **Real-time Feedback:**
        - Errors in the input (e.g., invalid regular expressions) are gracefully handled, with informative messages displayed in the terminal output area.
5. **Development Environment – Visual Studio Code (VSCode):**
    - **Code Management:**
        - Python scripts were developed, debugged, and executed using the VSCode IDE, which provides a robust environment for project organization.
    - **Version Control:**
        - Ensures maintainable and collaborative code through features like Git integration.

Additional Features and Educational Value

- **Interactive Learning:**
    - This project provides an accessible way for students to understand the abstract concept of Thompson's Construction by visualizing how a regular expression transforms into an NFA step-by-step.
- **Modular Code Design:**
    - Functions for different operations (e.g., apply_union, apply_kleene_star) make the code reusable and adaptable for further experimentation.
- **Error Handling:**
    - The system validates regular expressions and ensures appropriate error messages guide users through corrections.

Conclusion

This project successfully combines theoretical concepts from computation theory with practical programming skills. The use of Python and associated libraries showcases how computational models can be simulated, visualized, and understood interactively, making this an invaluable tool for both education and experimentation in the field of automata theory.

CODE AND EXPLANATION

```python
import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk
import graphviz

# Define State and NFA classes
class State:
    def __init__(self):
        self.transitions = {}
        self.is_accept = False

class NFA:
    def __init__(self, start, accept):
        self.start = start
        self.accept = accept

# Functions to build NFA components
def create_basic_nfa(char):
    start = State()
    accept = State()
    start.transitions[char] = [accept]
    return NFA(start, accept)

def apply_concatenation(nfa1, nfa2):
    nfa1.accept.transitions['ε'] = [nfa2.start]
    return NFA(nfa1.start, nfa2.accept)

def apply_union(nfa1, nfa2):
    start = State()
    accept = State()
    start.transitions['ε'] = [nfa1.start, nfa2.start]
    nfa1.accept.transitions['ε'] = [accept]
    nfa2.accept.transitions['ε'] = [accept]
    return NFA(start, accept)

def apply_kleene_star(nfa):
    start = State()
    accept = State()
    start.transitions['ε'] = [nfa.start, accept]
    nfa.accept.transitions['ε'] = [nfa.start, accept]
    return NFA(start, accept)

def regex_to_postfix(regex):
    precedence = {'*': 3, '.': 2, '|': 1}
    output = []
    operators = []
    chars = set("abcdefghijklmnopqrstuvwxyz0123456789")

    new_regex = []
    for i, char in enumerate(regex):
        new_regex.append(char)
        if i + 1 < len(regex) and (
            char in chars or char == ')' or char == '*') and (
            regex[i + 1] in chars or regex[i + 1] == '('):
            new_regex.append('.')
    regex = ''.join(new_regex)

    for char in regex:
        if char in chars:
            output.append(char)
        elif char == '(':
            operators.append(char)
        elif char == ')':
            while operators and operators[-1] != '(':
                output.append(operators.pop())
            operators.pop()
        elif char in precedence:
            while (operators and operators[-1] != '(' and
                   precedence[operators[-1]] >= precedence[char]):
                output.append(operators.pop())
            operators.append(char)
    while operators:
        output.append(operators.pop())

    return ''.join(output)

def thompsons_construction(regex):
    stack = []
    postfix = regex_to_postfix(regex)

    for char in postfix:
        if char.isalnum():
            stack.append(create_basic_nfa(char))
        elif char == '.':
            nfa2 = stack.pop()
            nfa1 = stack.pop()
            stack.append(apply_concatenation(nfa1, nfa2))
        elif char == '|':
            nfa2 = stack.pop()
            nfa1 = stack.pop()
            stack.append(apply_union(nfa1, nfa2))
        elif char == '*':
            nfa = stack.pop()
            stack.append(apply_kleene_star(nfa))
    if len(stack) != 1:
        raise ValueError("Invalid regular expression: remaining items on the stack after processing.")

    return stack[0]

def visualize_nfa(nfa, filename='nfa'):
    dot = graphviz.Digraph(format='png')
    dot.attr(rankdir='LR')  # Set horizontal layout
    dot.attr(dpi='300')  # Increase DPI for higher resolution
    state_ids = {}

    def get_state_id(state):
        if state not in state_ids:
            state_ids[state] = f's{len(state_ids)}'
        return state_ids[state]

    def add_state(state, is_initial=False, is_final=False):
        state_id = get_state_id(state)
        if is_initial:
            fillcolor = 'lightblue'
        elif is_final:
            fillcolor = 'lightblue'
        else:
            fillcolor = 'lightyellow'

        shape = 'doublecircle' if state.is_accept else 'circle'
        dot.node(state_id, shape=shape, style='filled', fillcolor=fillcolor)

        for char, next_states in state.transitions.items():
            for next_state in next_states:
                dot.edge(state_id, get_state_id(next_state), label=char)

    def traverse(state, visited, is_initial=False):
        if state in visited:
            return
        visited.add(state)
        add_state(state, is_initial=is_initial, is_final=state.is_accept)
        for next_states in state.transitions.values():
            for next_state in next_states:
                traverse(next_state, visited, is_initial=False)

    # Set the initial state as light blue
    traverse(nfa.start, set(), is_initial=True)
    dot.render(filename, cleanup=True)

# Tkinter GUI Code
def on_generate_click():
    regex = regex_entry.get()  # Get regex input from entry widget
    try:
        # Perform NFA generation and visualization
        nfa = thompsons_construction(regex)
        nfa.accept.is_accept = True
        visualize_nfa(nfa, 'nfa')

        # Display success message in terminal output
        terminal_output.insert(tk.END, f"Successfully generated NFA for regex: {regex}\n")

        # Load and display the NFA image
        img = Image.open('nfa.png')
        # Dynamically resize the image to fit in the GUI window
        window_width = root.winfo_width()
        window_height = root.winfo_height()
        max_width = int(window_width * 0.8)
        max_height = int(window_height * 0.6)
        img.thumbnail((max_width, max_height), Image.Resampling.LANCZOS)
        img = ImageTk.PhotoImage(img)
        img_label.config(image=img)
        img_label.image = img  # Keep a reference to prevent garbage collection

    except Exception as e:
        terminal_output.insert(tk.END, f"Error: {e}\n")

# Create Tkinter window
root = tk.Tk()
root.title("NFA Visualization")
root.geometry("1000x700")  # Initial window size

# Create Widgets
regex_label = tk.Label(root, text="Enter Regular Expression:")
regex_label.pack()

regex_entry = tk.Entry(root, width=40)
regex_entry.pack()

generate_button = tk.Button(root, text="Generate NFA", command=on_generate_click)
generate_button.pack()

terminal_output = tk.Text(root, height=10, width=50)
terminal_output.pack()

img_label = tk.Label(root)
img_label.pack()

# Run the Tkinter main loop
root.mainloop()
```

Code Explanation

1. State and NFA Classes

- **State Class:**
  - Represents a single state in the NFA.
  - Has a transitions dictionary to map input symbols to their target states.
  - Includes an *is_accept* flag to mark accepting (final) states.
- **NFA Class:**
  - Represents the complete NFA.
  - Stores references to the start and accept states of the NFA.

2. Building Blocks for NFA

- **create_basic_nfa(char):**
  Creates an NFA for a single input character. The NFA consists of a start and an accept state, with a transition labeled by the character between them.
- **apply_concatenation(nfa1, nfa2):**
  Connects two NFAs sequentially by linking the accept state of nfa1 to the start state of nfa2 using an ε-transition.
- **apply_union(nfa1, nfa2):**
  Combines two NFAs in parallel, creating a new start state and accept state, with ε-transitions connecting them to the start and accept states of both NFAs.
- **apply_kleene_star(nfa):**
  Implements the Kleene star operation by adding a new start and accept state. ε-transitions allow looping within the NFA or jumping directly to the new accept state.

3. Regular Expression Parsing

- **regex_to_postfix(regex):**
  - Converts an infix regular expression (e.g., a|b) into postfix notation (e.g., ab|), which is easier to process for Thompson's Construction.
  - Adds explicit concatenation (.) operators between adjacent characters, using operator precedence to maintain correct order.

4. Thompson's Construction

- **thompsons_construction(regex):**
  - Processes the postfix regular expression to build an NFA using a stack.
  - Pushes basic NFAs onto the stack for each character.
  - Pops and combines NFAs using the operations (concatenation, union, Kleene star) based on the current operator.
  - Returns the final NFA, with a single start and accept state.

5. Visualization

- **visualize_nfa(nfa, filename):**
  - Uses **Graphviz** to render the NFA as a directed graph.
  - Nodes represent states, styled as double circles for accepting states and circles for others.
  - Edges represent transitions, labeled with input symbols (or ε for epsilon transitions).

  - Traverses the NFA to add all states and transitions to the graph.

6. Tkinter GUI

- **GUI Components:**
  - **Input Widgets:** Users input the regular expression through a text entry field.
  - **Generate Button:** Triggers the NFA generation and visualization process.
  - **Terminal Output:** Displays success messages or errors during NFA generation.
  - **Image Display:** Dynamically resizes and displays the generated NFA graph.
- **on_generate_click():**
  - Fetches the user's input regular expression.
  - Generates the NFA using Thompson's construction.
  - Visualizes the NFA and renders it as an image file using Graphviz.
  - Resizes and displays the image in the GUI using **Pillow**.

Tools and Libraries Used

- **Tkinter:** Provides the graphical user interface for user interaction.
- **Pillow:** Handles resizing and displaying images in the GUI.
- **Graphviz:** Renders the NFA as a directed graph and outputs it as an image.
- **Python:** Implements the NFA construction logic and backend operations.

Workflow

1. The user enters a regular expression in the GUI.
2. The program converts the regex to postfix notation.
3. Using Thompson's construction, the program builds an NFA from the regex.
4. The NFA is visualized as a graph and displayed in the GUI, allowing the user to see the structure and transitions of the automaton.

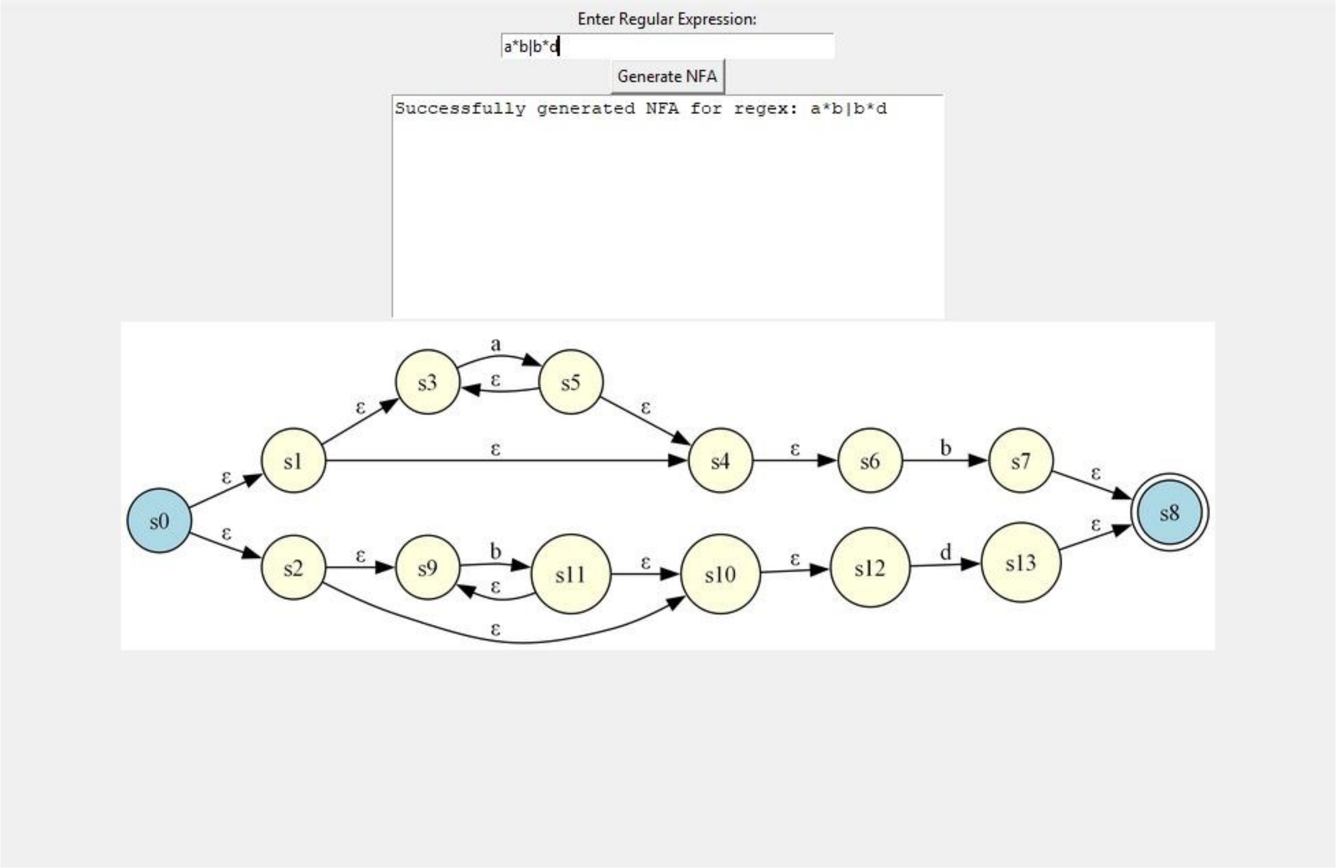
This code combines theoretical concepts of automata with practical Python programming to deliver an interactive learning tool for understanding regex-to-NFA conversions.

SNAPSHOTS

GPS Map Camera
Google
Bengaluru, Karnataka, India
41, Ags Colony, Anandnagar, Rt Nagar, Bengaluru,
Karnataka 560024, India
Lat 13.032679° Long 77.592843°
06/12/24 11:12 AM GMT +05:30

GPS Map Camera
Google
Bengaluru, Karnataka, India
41, Ags Colony, Anandnagar, Rt Nagar, Bengaluru,
Karnataka 560024, India
Lat 13.032667° Long 77.592792°
06/12/24 11:12 AM GMT +05:30

INPUT/OUTPUT WINDOW



Enter Regular Expression: a*b|b*d

```
Successfully generated NFA for regex: a*b|b*d
```