Table of Contents

Team Information

Team Name

orozcoaniceto

Team Members

- Letty Orozco
- Gustavo Aniceto

Time Spent

- **Total**: 12 hours
- **Letty Orozco**: 6 hours
- **Gustavo Aniceto**: 6 hours

Code Development and Testing

In undertaking this project, we committed to a rigorous development and testing methodology, emphasizing collaboration, quality, and continuous improvement. Our strategy was anchored around the use of GitHub, a decision that significantly streamlined our collaboration and code management processes.

Version Control with GitHub: GitHub's robust version control system was crucial for tracking changes, managing versions, and documenting our development process. We adopted a feature-branch workflow, where each member developed new features or bug fixes in isolated branches. This approach minimized conflicts during merge operations and ensured that our main branch always contained the most stable and reviewed version of our application.

Code Reviews: One of our core practices was conducting thorough code reviews. Every pull request initiated a code review process, requiring at least one team member to examine the changes. This practice fostered a culture of collective code ownership and quality, helping us to catch bugs, improve code readability, and share knowledge among team members.

Agile Methodology: We also embraced elements of the Agile methodology, with regular stand-ups to discuss progress, adapt to changes, and prioritize tasks. This flexible approach allowed us to respond quickly to new requirements or challenges that arose during development.

Programming Language and Libraries

- **Language**: Python 3.11.4
- **Libraries**:
  - `csv`: Essential for handling CSV file operations such as reading NFA definitions.
  - `collections`: Utilized for data structures like deque, which offered efficient append and pop operations for our algorithm implementations.
  - Standard libraries for file handling, regular expressions (`re`), and other basic functionalities.

Program Descriptions

Program 1: Tracing NFA Behavior

Key Data Structures

- **Stacks**: Stacks are crucial in this program, especially in the recursive function `_trace_nfa`, where the call stack implicitly acts as a stack to keep track of the current path in the NFA. This stack-based recursion facilitates depth-first search through the NFA, necessary for tracing all possible paths.

- **Deques**: While the direct implementation doesn't explicitly use deques, the Python list used in managing the current path and state transitions can benefit from deque's efficient append and pop operations, especially in large NFAs. This efficiency is critical in handling large sets of transitions and states during the trace.

- **Dictionaries**: The `trans_dict` in `NFATracer` is a dictionary data structure, where keys are state/input pairs and values are lists of resultant states. This structure is fundamental for efficiently storing and accessing the transitions of the NFA, allowing quick lookups during the tracing process.

Program Overview

This program takes a Non-deterministic Finite Automaton (NFA) and a string as input, tracing all possible paths and highlighting those ending in an accept state. The tracing algorithm employs depth-first search by utilizing the Python call stack for recursive function calls. This stack-based approach, combined with dictionary

usage for transition lookups, is essential for handling complex NFA structures, including those with ε transitions. Special attention is given to efficiently processing transitions and backtracking to explore every possible path.

Program Functionality

- **Initialization**: The `NFATracer` class is initialized with the filename of the NFA definition. The transitions of the NFA are read from a CSV file and stored in a dictionary (`trans_dict`), with the initial state(s) stored in a list (`start_state`).
- **String Tracing**: The `trace_string` method is used to trace a given string through the NFA. This method orchestrates the process, calling a recursive helper function to explore all paths.
- **Path Exploration**: The private method `_trace_nfa` performs the core tracing logic. It recursively follows each possible transition, using the call stack as an implicit stack data structure to track the current path and backtrack when necessary. The method handles both standard and ε transitions.
- **Output Generation**: The traced paths are categorized into accepting, rejecting, and trap states. This categorization and the paths are then outputted to a file, providing a clear and detailed trace of the string through the NFA.

Test Cases and Rationale

- **NFA N1**: (q1, q2, q3, q4) with epsilon transition from q2 to q3
    - **Test String**: 010110
        - **Purpose**: Tests multiple transitions, including a move from q1 to q2, epsilon transition from q2 to q3, and reaching the accepting state q4. It effectively challenges normal transitions, epsilon transitions, and concluding at an accept state.
    - **Additional Test Strings for N1**:
        - **String**: 111
            - **Purpose**: Tests the loop back to q1 on receiving 1, then transitions to q2 with a 1, checking if the NFA remains within q1 with consecutive 1s before moving to q2.
        - **String**: 010
            - **Purpose**: Tests transition from q1 to q2, then epsilon transition to q3, examining if q3 correctly doesn't lead to an accept state (*q4) when the input string ends.
        - **String**: ``
            - **Purpose**: An empty string checks the NFA's ability to handle no input, determining if it can identify or reject accept states from the initial state.
- **NFA N3**: (q1, q2, q3, q4, q5, q6) with epsilon transitions from q1 to *q2 and *q4
    - **Test Strings for N3**:

        - **String**: 00
            - **Purpose**: Verifies the epsilon transition functionality and the looping over q3 and q6 to respective accept states. Checks if the NFA can handle transitions to multiple states via epsilon transitions.
        - **String**: 000
            - **Purpose**: Further tests the looping mechanism and the NFA's ability to navigate states repeatedly, particularly verifying correct handling of q6 looping back to *q4.
        - **String**: ``
            - **Purpose**: Ensures the NFA correctly accepts an empty string, leading to an accept state via epsilon transitions.
- **NFA N4**: (q1, q2, q3) with epsilon transition from *q1 to q3
    - **Test Strings for N4**:
        - **String**: baa
            - **Purpose**: Checks transitioning from *q1 to q2 with b, looping in q2 with a, and moving to q3 with another a. Validates the NFA's handling of different inputs and state transitions, including accept state arrival.
        - **String**: bb
            - **Purpose**: Tests the NFA's management of undefined transitions (e.g., q2 to q3 with b), exploring how the NFA deals with inputs leading to uncharted transitions.
        - **String**: ``
            - **Purpose**: Similar to other NFAs, this string evaluates the handling of an empty string, focusing on correct processing of epsilon transitions to an accept state.

Each test string is carefully selected to challenge specific aspects of the NFAs, like looping, state transitions (including epsilon transitions), and accurate identification of accept states, ensuring thorough validation and robustness.

Program 2: NFA to DFA Conversion and Minimization

This program encompasses two main components: converting a Non-deterministic Finite Automaton (NFA) to a Deterministic Finite Automaton (DFA) and then minimizing this DFA. The `NFAToDFAConverter` class manages the conversion process, transforming the NFA's states and transitions into a DFA format. The `DFAMinimizer` class then takes this DFA and minimizes its states, aiming for an equivalent DFA with fewer states.

Key Data Structures

- `NFAToDFAConverter` **Class**:

  - `self.nfa_transitions`: A `defaultdict(list)` storing the NFA transitions, facilitating dynamic state and transition additions.
  - `self.start_state`: Records the initial state of the NFA.
  - `self.accept_states`: A set maintaining the accepting states of the NFA.

- `DFAMinimizer` **Class**:

  - `self.dfa`: Similar to `nfa_transitions` in `NFAToDFAConverter`, this `defaultdict(list)` holds the DFA transitions.
  - `self.start_state` and `self.accept_states`: Store the start and accept states of the DFA, respectively.

Program Overview

The `NFAToDFAConverter` class converts an NFA into a DFA by analyzing and restructuring the state transitions of the NFA into a deterministic format. The conversion involves handling multiple transitions for the same symbol from a single state, incorporating epsilon transitions, and identifying the new accept states in the resultant DFA.

Following the conversion, the `DFAMinimizer` class takes over to reduce the DFA's state count. This minimization is vital to simplify the DFA without losing its language recognition capability. The class identifies equivalent states and merges them, effectively reducing the overall state count.

Program Functionality

- `NFAToDFAConverter`:

  - Converts NFA transitions to a DFA format.
  - Handles epsilon (~) transitions and merges states where necessary.
  - Identifies all possible DFA accept states originating from the NFA's accept states.

- `DFAMinimizer`:

  - Executes a breadth-first search (BFS) to find all reachable states from the DFA's start state.
  - Partitions DFA states into groups of equivalent states and minimizes the DFA by merging these groups.
  - Ensures the minimized DFA accurately recognizes the same language as the original DFA.

Test Cases and Rationale

- **NFAToDFAConverter**:

  - **Simple NFAs**: Test NFAs without epsilon transitions to ensure straightforward conversion is handled correctly.
  - **NFAs with Epsilon Transitions**: Check how well the converter handles epsilon closures and transitions.
  - **Overlapping Transitions**: Test NFAs where multiple transition paths exist for the same input symbol in a single state.
  - **Complex NFAs**: Utilize NFAs with multiple accept states to verify accurate state and accept state identification in the resultant DFA.

- **DFAMinimizer**:
    - **Redundant State Removal**: Ensure the minimizer successfully identifies and removes states not contributing to the DFA's language recognition.
    - **Reachability**: Confirm all states in the minimized DFA are reachable from the start state.

Conclusion

The `NFAToDFAConverter` and `DFAMinimizer` classes, along with their associated test cases, provide a comprehensive approach to understanding and manipulating finite automata. The tests are designed to cover a range of scenarios, from basic to complex NFAs, ensuring robust conversion and minimization processes. The program's functionality focuses on accurate conversion and effective minimization, emphasizing language preservation and state efficiency.

Extra Credit Content

Program 3: Regular Expression to NFA Converter

This program focuses on converting regular expressions into Non-deterministic Finite Automatons (NFAs). It encapsulates the complexity of regular expressions in a graphical form, making them easier to understand and manipulate.

Key Data Structures

- **States (`self.states`)**:
    - **Type**: `set`
    - **Usage**: Stores all the states of the NFA. Each state is a unique string (e.g., "q0", "q1"). The usage of a set ensures that each state is stored only once and can be efficiently checked for existence.
- **Transitions (`self.transitions`)**:
    - **Type**: `list` of tuples
    - **Usage**: Each tuple represents a transition and contains `(start_state, input_symbol, end_state)`. The list structure allows for easy addition of transitions and iteration over them.
- **Alphabet (`self.alphabet`)**:
    - **Type**: `set`
    - **Usage**: Represents the set of input symbols (characters) that the NFA can process, excluding the epsilon (~) character.
- **Final States (`self.final_states`)**:
    - **Type**: `set`
    - **Usage**: Keeps track of accept states. Using a set allows for quick checks on whether a state is an accept state.
- **Start State (`self.start_state`)**:
    - **Type**: `string`

    - **Usage**: Holds the initial state of the NFA.

- **State Counter (`self.state_counter`)**:
    - **Type**: `int`
    - **Usage**: A counter to ensure unique state names are generated dynamically.

Test Cases and Reasoning

For the regex2nfa converter, test cases are essential to ensure the accurate handling of various regex constructs. The tests cover a range of patterns, from basic concatenation to more complex nested expressions.

- **Simple Concatenation** (e.g., "ab"):
    - Validates basic sequencing of characters.
- **Union Operator** (e.g., "aUb"):
    - Tests the NFA's ability to handle choices or alternatives between different sequences.
- **Kleene Star** (e.g., "a*"):
    - Ensures that the NFA can handle repetition, including zero occurrences of a character.
- **Nested Expressions** (e.g., "(aUb)c"):
    - Tests the NFA's capability to handle complex expressions involving nested operations.
- **Combination of Operators** (e.g., "(aUb)*c"):
    - Evaluates the handling of combined regex operators in a single expression.
- **Empty String and Epsilon Transitions** (e.g., "a~b" for "ab" with an explicit epsilon):
    - Verifies how the NFA deals with epsilon transitions and empty strings.
- **Edge Cases** (e.g., "", "a", "(aUb)*"):
    - Examines the resilience of the NFA builder against irregular or malformed regex strings, including empty regex or improper nesting and repetition symbols.

Validation Methodology

The validation of the constructed NFA includes:

- **Trace Matches**:
    - Generate strings that should match and shouldn't match each test regex. Verify the NFA's acceptance or rejection of these strings.
- **Compare with Theoretical or Established Models**:
    - Use theoretical expectations of NFA constructions from textbooks or literature for comparison.
- **Visual Inspection**:

- Manually inspect the NFA structure and transitions for smaller or simpler regex expressions.

Conclusion

The regex2nfa converter's testing and validation process ensures that the NFAs it generates are accurate representations of their corresponding regular expressions. The thorough test cases range from simple concatenations to complex nested and combined expressions. This comprehensive approach confirms the converter's capability to handle a wide spectrum of regular expressions, thereby validating its effectiveness and reliability.