Hajin Park A18596632
CSE 105 Winter 2025
Project Task 1
March 15, 2025

---

1 Design Choices and Implementation Overview

I chose to implement this program in Python because of its familiarity, readability, and its powerful built-in string parsing abilities:

- **parse_dfa**: Parses and validates a multi-line string representing a DFA. It ensures that all required definitions (states, alphabet, start, accept, transitions) are present and correctly formatted. The function checks that:
  - The `states`, `alphabet`, `start`, and `accept` fields are non-empty.
  - The start state is included in the set of states.
  - All accept states belong to the set of states.
  - Each transition is well-formed, and uses states and symbols defined in the DFA.
- **construct_intersection_dfa**: Implements the Cartesian product construction to combine two DFAs. The resulting DFA accepts a string if and only if it is accepted by both input DFAs.
- **is_language_empty**: Uses a breadth-first search (BFS) to check whether the intersection DFA has any reachable accepting states.
- **are_properties_consistent**: Executes the previous functions to determine whether the languages of the two DFAs have a non-empty intersection.

2 DFA Parsing and Representation

Input strings representing DFAs follow the format:

```
states: q0,q1,q2
alphabet: 0,1
start: q0
accept: q1
transitions:
q0,0,q1
q0,1,q2
q1,0,q1
q1,1,q1
q2,0,q2
q2,1,q2
```

The parse_dfa function processes each lines and validates:

- The presence of all required definitions.
- Non-empty values for states, alphabet, start, and accept.
- Correctness of transitions (each transition must have exactly three comma-separated fields; the current state and next state must be in the states set, and the symbol must belong to the alphabet).
- w_0: Represents a DFA that accepts strings starting with `0`.
- w_1: Represents a DFA that accepts strings starting with `1`.
- x_0: Represents a DFA that accepts strings containing at least one `0`.
- x_1: Represents a DFA that accepts strings containing at least one `1`.

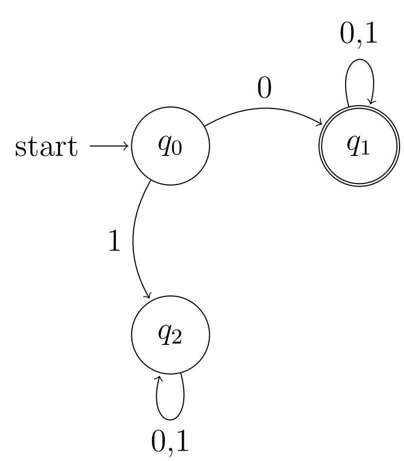
1. DFA for Strings Starting with 0 (w_0)

Formal Definition: Let

$$M_0 = (Q, \Sigma, \delta, q_0, F)$$

where:

$$\begin{aligned}
Q &= \{q_0, q_1, q_2\}, \\
\Sigma &= \{0, 1\}, \\
q_0 &\text{ is the start state,} \\
F &= \{q_1\}, \\
\delta(q_0, 0) = q_1, &\quad \delta(q_0, 1) = q_2, \\
\delta(q_1, 0) = q_1, &\quad \delta(q_1, 1) = q_1, \\
\delta(q_2, 0) = q_2, &\quad \delta(q_2, 1) = q_2.
\end{aligned}$$



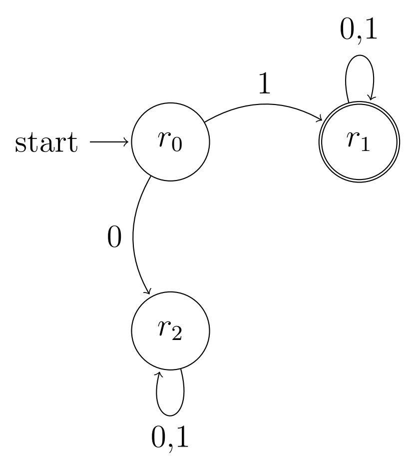
2. DFA for Strings Starting with 1 (w_1)

Formal Definition: Let

$$M_1 = (Q, \Sigma, \delta, r_0, F)$$

where:

$$\begin{aligned}
Q &= \{r_0, r_1, r_2\}, \\
\Sigma &= \{0, 1\}, \\
&r_0 \text{ is the start state}, \\
F &= \{r_1\}, \\
\delta(r_0, 1) = r_1, &\quad \delta(r_0, 0) = r_2, \\
\delta(r_1, 0) = r_1, &\quad \delta(r_1, 1) = r_1, \\
\delta(r_2, 0) = r_2, &\quad \delta(r_2, 1) = r_2.
\end{aligned}$$



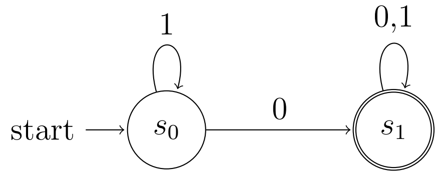
3. DFA for Strings Containing at Least One 0 (x_0)

Formal Definition: Let

$$M_2 = (Q, \Sigma, \delta, s_0, F)$$

where:

$$\begin{aligned}
Q &= \{s_0, s_1\}, \\
\Sigma &= \{0, 1\}, \\
&s_0 \text{ is the start state}, \\
F &= \{s_1\}, \\
\delta(s_0, 0) = s_1, &\quad \delta(s_0, 1) = s_0, \\
\delta(s_1, 0) = s_1, &\quad \delta(s_1, 1) = s_1.
\end{aligned}$$



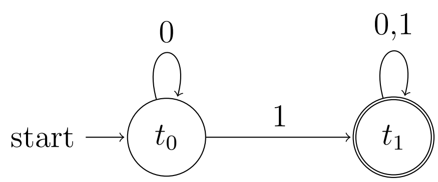
4. DFA for Strings Containing at Least One 1 (x_1)

Formal Definition: Let

$$M_3 = (Q, \Sigma, \delta, t_0, F)$$

where:

$$\begin{aligned}
Q &= \{t_0, t_1\}, \\
\Sigma &= \{0, 1\}, \\
t_0 &\text{ is the start state}, \\
F &= \{t_1\}, \\
\delta(t_0, 1) = t_1, &\quad \delta(t_0, 0) = t_0, \\
\delta(t_1, 0) = t_1, &\quad \delta(t_1, 1) = t_1.
\end{aligned}$$



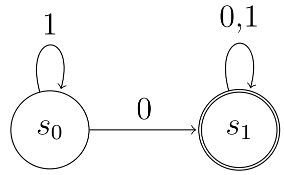
5. DFA with Missing Start Value (y0)

Formal Definition: Let

$$M_{y0} = (Q, \Sigma, \delta, q_0, F)$$

with:

$$\begin{aligned}
Q &= \{s_0, s_1\}, \\
\Sigma &= \{0, 1\}, \\
q_0 &\textbf{ is not provided (missing)}, \\
F &= \{s_1\}, \\
\delta(s_0, 0) = s_1, &\quad \delta(s_0, 1) = s_0, \\
\delta(s_1, 0) = s_1, &\quad \delta(s_1, 1) = s_1.
\end{aligned}$$



6. DFA with Missing States (y1)

Formal Definition: The encoding for this DFA fails to define its state set.

$$M_{y1} = (Q, \Sigma, \delta, q_0, F)$$

$$\begin{aligned} Q &= \{\}, \\ \Sigma &= \{0, 1\}, \\ q_0 &= t_0, \\ F &= \{t_1\}, \\ \delta(t_0, 1) = t_1, &\quad \delta(t_0, 0) = t_0, \\ \delta(t_1, 0) = t_1, &\quad \delta(t_1, 1) = t_1. \end{aligned}$$

7. DFA with Incomplete Transition (y2)

Formal Definition: Let

$$M_{y2} = (Q, \Sigma, \delta, q_0, F)$$

where:

$$\begin{aligned} Q &= \{a, b\}, \\ \Sigma &= \{0, 1\}, \\ q_0 &= a, \\ F &= \{b\}, \\ \delta(a, 0) &= b, \\ \delta(a, 1) \end{aligned}$$

3 Code

```
"""
Project: CSE105W25 Task 1
Author: Hajin Park
Date: 3/15/2025
"""

from collections import deque
```

```
def parse_dfa(dfa_str):
    """
    Parses a DFA from a string representation and validates its format.

    Expected format:
        states: state1,state2,...
        alphabet: symbol1,symbol2,...
        start: start_state
        accept: accept_state1,accept_state2,...
        transitions:
        current_state,symbol,next_state
        ...

    Validations performed:
      - All required headers must be present.
      - None of the required fields (states, alphabet, start, accept) may be
         empty.
      - The start state must be in the set of states.
      - All accept states must be in the set of states.
      - Each transition line must consist of exactly three comma-separated
         values.
      - For each transition, the current state and next state must belong to
         the set of states,
        and the symbol must be in the alphabet.

    Returns:
        A dictionary with the following keys:
          - 'states': set of states
          - 'alphabet': set of symbols
          - 'start': start state
          - 'accept': set of accepting states
          - 'transitions': dictionary mapping (state, symbol) -> next_state

    Raises:
        ValueError: If the DFA string is not correctly encoded.
    """
    lines = [line.strip() for line in dfa_str.strip().splitlines() if line.
       strip()]
    dfa = {}
    transitions = {}

    # Track whether each header has been found.
    headers_found = {
        "states": False,
        "alphabet": False,
        "start": False,
        "accept": False,
        "transitions": False,
    }

    for i, line in enumerate(lines):
        if line.startswith("states:"):
            headers_found["states"] = True
            states_val = line[len("states:") :].strip()
```

```
                if not states_val:
                    raise ValueError("The 'states' line is empty.")
                dfa["states"] = set(s.strip() for s in states_val.split(",") if s.
                    strip())
            elif line.startswith("alphabet:"):
                headers_found["alphabet"] = True
                alphabet_val = line[len("alphabet:") :].strip()
                if not alphabet_val:
                    raise ValueError("The 'alphabet' line is empty.")
                dfa["alphabet"] = set(
                    s.strip() for s in alphabet_val.split(",") if s.strip()
                )
            elif line.startswith("start:"):
                headers_found["start"] = True
                start_val = line[len("start:") :].strip()
                if not start_val:
                    raise ValueError("The 'start' state is empty.")
                dfa["start"] = start_val
            elif line.startswith("accept:"):
                headers_found["accept"] = True
                accept_val = line[len("accept:") :].strip()
                if not accept_val:
                    raise ValueError("The 'accept' states line is empty.")
                dfa["accept"] = set(s.strip() for s in accept_val.split(",") if s.
                    strip())
            elif line.startswith("transitions:"):
                headers_found["transitions"] = True
                # Process the rest of the lines as transitions.
                for trans_line in lines[i + 1 :]:
                    parts = [p.strip() for p in trans_line.split(",")]
                    if len(parts) != 3:
                        raise ValueError(f"Incomplete transition line: '{trans_line
                            }'")
                    curr, sym, nxt = parts
                    transitions[(curr, sym)] = nxt
                break  # Exit once transitions are processed.

        # Ensure all required headers were found.
        for key, found in headers_found.items():
            if not found:
                raise ValueError(f"Missing required DFA component: {key}")

        dfa["transitions"] = transitions

        # Validate that the start state is in the set of states.
        if dfa["start"] not in dfa["states"]:
            raise ValueError("The start state is not in the set of states.")
        # Validate that each accept state is in the set of states.
        if not dfa["accept"].issubset(dfa["states"]):
            raise ValueError("Some accept states are not in the set of states.")
        # Validate each transition.
        for (curr, sym), nxt in transitions.items():
            if curr not in dfa["states"]:
                raise ValueError(f"Transition error: current state '{curr}' not in
                    states.")
            if nxt not in dfa["states"]:
```

```
            raise ValueError(f"Transition␣error:␣next␣state␣'{nxt}'␣not␣in␣states.")
        if sym not in dfa["alphabet"]:
            raise ValueError(f"Transition␣error:␣symbol␣'{sym}'␣not␣in␣alphabet.")

    return dfa


def construct_intersection_dfa(dfa1, dfa2):
    """
    Constructs the intersection DFA of two DFAs using the Cartesian product
        construction.

    Assumes both DFAs share a common alphabet (the intersection of their
        alphabets is used).

    Returns:
        A dictionary representing the intersection DFA with keys:
          'states', 'alphabet', 'start', 'accept', and 'transitions'.
    """
    common_alphabet = dfa1["alphabet"] & dfa2["alphabet"]
    if not common_alphabet:
        raise ValueError("The␣two␣DFAs␣do␣not␣share␣a␣common␣alphabet.")

    new_states = set()
    new_transitions = {}
    new_accept = set()

    start_state = (dfa1["start"], dfa2["start"])
    queue = deque([start_state])
    new_states.add(start_state)

    while queue:
        (s1, s2) = queue.popleft()
        # Mark a state as accepting if both components are accepting.
        if s1 in dfa1["accept"] and s2 in dfa2["accept"]:
            new_accept.add((s1, s2))
        for sym in common_alphabet:
            next1 = dfa1["transitions"].get((s1, sym))
            next2 = dfa2["transitions"].get((s2, sym))
            if next1 is None or next2 is None:
                continue  # Skip if any DFA has no transition for this symbol.
            next_state = (next1, next2)
            new_transitions[((s1, s2), sym)] = next_state
            if next_state not in new_states:
                new_states.add(next_state)
                queue.append(next_state)

    return {
        "states": new_states,
        "alphabet": common_alphabet,
        "start": start_state,
        "accept": new_accept,
        "transitions": new_transitions,
    }
```

```python
def is_language_empty(dfa):
    """
    Determines whether the language recognized by a DFA is empty.

    Uses Breadth-First Search (BFS) from the DFA's start state to see if any
        accepting state is reachable.

    Returns:
        True if the language is empty (no accepting state is reachable);
            otherwise, False.
    """
    visited = set()
    queue = deque([dfa["start"]])

    while queue:
        state = queue.popleft()
        if state in dfa["accept"]:
            return False  # An accepting state is reachable.
        if state in visited:
            continue
        visited.add(state)
        for sym in dfa["alphabet"]:
            next_state = dfa["transitions"].get((state, sym))
            if next_state and next_state not in visited:
                queue.append(next_state)
    return True  # An accepting state is not reachable.


def are_properties_consistent(dfa_str1, dfa_str2):
    """
    Given two DFA representations as strings, determines whether their
        languages are consistent

    Returns:
        True if the intersection is nonempty (properties are consistent), else
            False.
    """
    dfa1 = parse_dfa(dfa_str1)
    dfa2 = parse_dfa(dfa_str2)
    intersection_dfa = construct_intersection_dfa(dfa1, dfa2)
    return not is_language_empty(intersection_dfa)

    ...
```

4 Demonstration Tests

Testing is performed using PyTest for ease of development and demonstration purposes.

Valid Cases

- **Consistency Test:** The DFAs w_0 and w_1 (strings starting with `0` and `1`) should be inconsistent.
- **Consistency Test:** The DFAs x_0 and x_1 (strings containing at least one `0` and at least one `1`) should be consistent.

Invalid Cases

The following invalid encodings are tested to confirm that the program correctly rejects them by raising a `ValueError`:

- A DFA with a missing start value.
- A DFA with an empty states field.
- A DFA with a incomplete transition (not exactly three comma-separated values).

```
import pytest
from main import are_properties_consistent

# Valid DFA Encodings

# DFA for strings that start with '0'
w0 = """\
states: q0,q1,q2
alphabet: 0,1
start: q0
accept: q1
transitions:
q0,0,q1
q0,1,q2
q1,0,q1
q1,1,q1
q2,0,q2
q2,1,q2
"""

# DFA for strings that start with '1'
w1 = """\
states: r0,r1,r2
alphabet: 0,1
start: r0
accept: r1
transitions:
r0,1,r1
r0,0,r2
r1,0,r1
r1,1,r1
r2,0,r2
r2,1,r2
"""

# DFA for strings that contain at least one '0'
x0 = """\
```

```
states: s0,s1
alphabet: 0,1
start: s0
accept: s1
transitions:
s0,0,s1
s0,1,s0
s1,0,s1
s1,1,s1
"""

# DFA for strings that contain at least one '1'
x1 = """\
states: t0,t1
alphabet: 0,1
start: t0
accept: t1
transitions:
t0,1,t1
t0,0,t0
t1,0,t1
t1,1,t1
"""

# Invalid DFA Encodings

# y0: Missing start value
y0 = """\
states: s0,s1
alphabet: 0,1
start:
accept: s1
transitions:
s0,0,s1
s0,1,s0
s1,0,s1
s1,1,s1
"""

# y1: Missing states
y1 = """\
states:
alphabet: 0,1
start: t0
accept: t1
transitions:
t0,1,t1
t0,0,t0
t1,0,t1
t1,1,t1
"""

# y2: Incomplete transition
y2 = """\
states: a,b
alphabet: 0,1
```

```
start: a
accept: b
transitions:
a,0,b
a,1  # Incomplete transition (missing next state)
"""

@pytest.mark.parametrize(
    "dfa_str1, dfa_str2, expected_error",
    [
        (y0, w0, "start"),                  # y0 is missing a valid start state
            .
        (w0, y1, "states"),                 # y1 is missing states.
        (y0, y1, "start"),                  # Either error could be raised; '
            start' error is acceptable.
        (y2, w0, "Incomplete transition"),  # y2 has a Incomplete transition.
    ]
)
def test_invalid_dfa_encoding(dfa_str1, dfa_str2, expected_error):
    """
    Test that passing incorrectly encoded DFA strings raises a ValueError.
    The error message should mention the missing or incomplete component.
    """
    with pytest.raises(ValueError, match=expected_error):
        are_properties_consistent(dfa_str1, dfa_str2)

def test_are_properties_consistent_inconsistent():
    """
    Test that DFAs for strings starting with '0' and '1' are inconsistent.
    (They cannot have any common accepted string.)
    """
    result = are_properties_consistent(w0, w1)
    assert result is False, "Expected inconsistency for DFAs that start with 
        '0' vs. '1'."

def test_are_properties_consistent_consistent():
    """
    Test that DFAs for strings that contain at least one '0' and at least one
        '1' are consistent.
    (The intersection accepts strings containing both '0' and '1'.)
    """
    result = are_properties_consistent(x0, x1)
    assert result is True, "Expected consistency for DFAs that require at least
        one '0' and one '1'."
```