Legal Document Clause Review Order

1 Problem Statement

You are building a document review system for Indian legal contracts and agreements. Legal documents often contain clauses that reference other clauses. For example, Clause 2.1 might say "as defined in Clause 1.3", creating a dependency where Clause 2.1 depends on Clause 1.3.

When reviewing a legal document, clauses must be reviewed in an order where all dependencies are satisfied—that is, if Clause A depends on Clause B, then Clause B must be reviewed before Clause A.

Given a set of clause dependencies, determine a valid review order. If the dependencies form a circular reference (e.g., Clause 2.1 depends on Clause 1.3, and Clause 1.3 depends on Clause 2.1), detect and report the cycle.

Context: In Indian legal practice, contracts often have complex inter-clause dependencies. A circular dependency indicates a drafting error that needs to be flagged. The review order helps lawyers understand clauses in the correct sequence.

1.1 Input Format

```
{
  "edges": [
    {"from": "1.3", "to": "2.1"},
    {"from": "2.1", "to": "3.2"}
  ]
}
```

Each edge represents a dependency: {`"from": "A", "to": "B"`} means "Clause B depends on Clause A" (Clause B references Clause A, so A must be reviewed before B).

Clause identifiers are strings (e.g., "1.3", "2.1", "Section_3_Subsection_2", etc.).

1.2 Output Format

If valid order exists:

```
{
  "order": ["1.3", "2.1", "3.2"]
}
```

The order array lists clauses in a valid review sequence where all dependencies are satisfied.

If circular dependency detected:

```
{
  "cycle": ["2.1", "1.3"]
}
```

The cycle array contains two clause identifiers representing the first circular dependency pair found (where one depends on the other and vice versa).

2 Example Test Cases

2.1 Test Case 1: Simple Dependency Chain

- **Input:** `["from": "1.1", "to": "2.1"]`
- **Output:** `{"order": ["1.1", "2.1"]}`
- **Explanation:** Clause 2.1 depends on Clause 1.1, so review 1.1 first

2.2 Test Case 2: Multiple Dependencies

- **Input:** `["from": "1.1", "to": "2.1", {"from": "1.2", "to": "2.1"}]`
- **Output:** `{"order": ["1.1", "1.2", "2.1"]}` (or `["1.2", "1.1", "2.1"]`)
- **Explanation:** Clause 2.1 depends on both 1.1 and 1.2. When multiple valid orderings exist, use lexicographic tie-breaking (alphabetical/numerical order)

2.3 Test Case 3: Circular Dependency Detected

- **Input:** `["from": "1.1", "to": "2.1", {"from": "2.1", "to": "1.1"}]`
- **Output:** `{"cycle": ["1.1", "2.1"]}` or `{"cycle": ["2.1", "1.1"]}`
- **Explanation:** Circular reference detected—Clause 1.1 depends on 2.1, and 2.1 depends on 1.1

2.4 Test Case 4: Complex Dependency Graph

- **Input:** `["from": "1.1", "to": "3.1", {"from": "2.1", "to": "3.1"}, {"from": "2.1", "to": "3.2"}]`
- **Output:** `{"order": ["1.1", "2.1", "3.1", "3.2"]}` (or similar valid ordering)
- **Explanation:** Clause 3.1 depends on both 1.1 and 2.1; Clause 3.2 depends on 2.1

2.5 Test Case 5: Self-Referencing Clause

- **Input:** `["from": "2.1", "to": "2.1"]`
- **Output:** `{"cycle": ["2.1", "2.1"]}`
- **Explanation:** A clause depending on itself is a circular dependency

3 Edge Cases

- Empty edges list → return order of all clauses mentioned (if any) in lexicographic order
- Self-loop (clause depends on itself) → detect as cycle
- Disconnected components (multiple independent clause groups) → valid ordering across all components
- Multiple valid orderings → use lexicographic tie-breaking for stable output
- Clauses mentioned only in "from" field (no dependencies) → include them in order

4 Requirements

- If no cycle exists, return a valid topological order where all dependencies are satisfied
- If a cycle exists, detect and report it with the first back-edge pair found
- When multiple valid orderings exist, use lexicographic (alphabetical/numerical) ordering for tie-breaking
- Include all clauses mentioned in the dependency graph in the output